Universidad
Carlos III de Madrid

Escuela Politécnica Superior

PROYECTO FIN DE GRADO

DESARROLLO DEL SISTEMA DE TELEOPERACIÓN DEL BRAZO DEL ROBOT HUMANOIDE RH2 UTILIZANDO KINECT

Autor: Iván Solano García

Tutor: Juan Miguel García Haro

Titulación: Grado en Ingeniería en Tecnologías Industriales

Convocatoria: Octubre 2014

Universidad
Carlos III de Madrid

Universidad Carlos III de Madrid

Agradecimientos

En primer lugar, agradecer a Juan Miguel García Haro que me ofreciese la posibilidad de realizar este proyecto, me facilitase todo el material que necesitaba y que le iba solicitando y que me ayudase a decidir qué camino seguir cuando estaba atascado.

También quiero mencionar a Pedro Portalatin Ginés, el cual me ayudo al principio del proyecto cuando tuve que montar el dispositivo Kinect y gracias al cual aprendí a soldar cables correctamente, a Santiago Morante, el cual me ayudó mucho a la hora de trabajar con Linux y, por último, a Juan G. Víctores, que siempre estaba dispuesto a ofrecerme su ayuda cuando tenía algún problema y no sabía cómo solucionarlo.

Finalmente agradecer a mi familia y amigos el apoyo recibido en momentos en los que no me encontraba con fuerzas para seguir, y ellos siempre encontraban la forma de animarme a continuar.

GRACIAS A TODOS

Universidad
Carlos III de Madrid

Universidad Carlos III de Madrid

Resumen

La finalidad de este proyecto consiste en realizar la teleoperación del brazo del robot humanoide TEO mediante el dispositivo Kinect, aunque finalmente por falta de tiempo no se pudo implementar en el robot. Para ello se partirá de un trabajo anterior realizado por Lorenzo Moral Durán y el cual se ampliará para conseguir dicha teleoperación en un espacio tridimensional programando un método para calcular las coordenadas articulares conocido como RPY (Roll Pitch Yaw) a partir de las coordenadas cartesianas de base del cuello, hombro, codo y muñeca.

Para ello nos serviremos de herramientas como Kinect for Windows SDK que nos proporcionará algunos ejemplos de esqueletización ya programados, Visual Studio para la programación en C++, OpenRave para la simulación del programa antes de utilizarlo en el propio robot o YARP para comunicar el ordenador donde se ejecuta el programa con el ordenador donde se ejecuta el simulador.

Se decidió usar las herramientas de software anteriormente citadas ya que eran las mismas que se habían utilizado en el proyecto anteriormente citado y, por lo tanto, estaba comprobado que nos ofrecían las herramientas necesarias para realizar el proyecto.

Palabras clave

C++, RPY, esqueletización, Kinect, Kinect for Windows SDK, OpenRAVE, robot humanoide, teleoperación, TEO, Visual Studio, YARP.

Universidad Carlos III de Madrid

Abstract

The main goal of this project is to do the teleoperation of the humanoid robot TEO's arm through the Kinect device. For this we are going to start from another project written by Lorenzo Moral Durán and which will be expanded to get the teleoperation aforementioned but in a three-dimensional space by programming a method to calculate the articular coordinates known as RPY (Roll Pitch Yaw) from the Cartesian coordinates of the neck base, shoulder, elbow and wrist.

For this we will use tools such as Kinect for Windows SDK which will give us some examples about skeletisation already programmed, Visual Studio for C++ programming, OpenRAVE for the simulation of the program before use it in the real robot or YARP to communicate the computer that runs the program and the computer that runs the simulator.

We decided to use the software tools aforementioned because they were the same tools used in Lorenzo's project and, therefore, it was checked that they offered the tools we needed to do the project.

Keywords

C++, RPY, eskeletisation, Kinect, Kinect for Windows SDK, OpenRAVE, humanoid robot, teleoperration, TEO, Visual Studio, YARP.

Universidad Carlos III de Madrid

ÍNDICE

Universidad Carlos III de Madrid

Índice de figuras

Universidad Carlos III de Madrid

1. Introducción

1.1. Motivación y contexto del proyecto

El objeto de este proyecto consiste en mostrar las posibilidades que un dispositivo asequible como es Kinect nos permite a la hora de controlar un robot de cualquier tipo únicamente a través del movimiento de nuestro cuerpo. Además el proyecto está realizado con software libre lo que disminuye su coste y facilita su implementación y desarrollo a cualquier persona.

Como se menciona anteriormente este proyecto parte de la base de otro realizado anteriormente pero en esta ocasión la teleoperación se realizara en tres dimensiones a diferencia del anterior, realizado en dos dimensiones, lo que amplia mucho su s posibilidades de aplicación práctica.

1.2. Objetivos

El objetivo principal de este proyecto consiste en la teleoperación del brazo del robot humanoide TEO utilizando para ello el dispositivo Kinect. Se busca que el movimiento se realice en tres dimensiones y que el robot reproduzca fielmente los movimientos realizados por la persona que se encuentra frente al dispositivo Kinect. Para llegar a alcanzar dicho objetivo se dividió el proceso en una serie de subobjetivos que se explicarán a continuación:

- Detectar una persona mediante el dispositivo Kinect y realizar la esqueletización de la misma.
- Cálculo de los ángulos articulares necesarios para que el robot ejecute los movimientos.
- Conexión del ordenador en el que se ejecuta el programa principal con el robot o con el ordenador en el que se ejecute el simulador.
- Pasar los ángulos articulares calculados al robot o al simulador, mostrándolos en este último caso por pantalla.
- Usar un simulador para comprobar el correcto funcionamiento del programa antes de implementarlo en el robot real.

Una vez alcanzados estos objetivos el último sería implementar el programa en el robot real y comprobar que funciona correctamente.

2. Teleoperación

2.1. Historia

La teleoperación nació junto con la industria nuclear debido a la necesidad de manejar materiales altamente radioactivos, muy peligrosos para la salud simplemente con estar en su presencia.

En 1945, la compañía Central Research Laboratories fue contratada para desarrollar un manipulador remoto para Argonne National Laboratory. El resultado fue el Manipulador Maestro-Esclavo Mk.8 o MSM-8, el cual se convirtió en un icono de la manipulación remota. Dicho dispositivo consistía en un manipulador (esclavo) que imitaba los movimientos que de un manipulador maestro controlado por un operario.

En 1954 Raymond Goertz creó el E1, el primer sistema de teleoperación accionado por electricidad y con servo controles tanto en el maestro como en el esclavo.

En los años setenta la teleoperación alcanzó su madurez con su aplicación en misiones espaciales, concretamente en los vehículos a control remoto Lunojod I y Lunojod II enviados a la Luna en 1970 y 1973 respectivamente.

En las últimas décadas, los avances en teleoperación han ido muy ligados a la evolución de la robótica y la informática. Gracias a estos avances, muchos sistemas de teleoperación actuales consisten en robots que tienen una gran autonomía y solo precisan ser teleoperados para determinadas acciones que, debido a las limitaciones de la robótica, no puede realizar por si solos. También se ha progresado en las interfaces hombre-máquina buscando una mayor sensación de control de la máquina y de telepresencia.

2.2. Elementos de un sistema teleoperado

Los sistemas de teleoperación están constituidos, en general, por dos manipuladores robóticos (uno local y otro remoto), un canal de comunicación, el medio ambiente con el que interactúa el robot remoto, y el operador humano.

Por lo tanto, un sistema de teleoperación se compone básicamente de un robot maestro controlado por un operador y un robot remoto, o esclavo, que se encarga de interaccionar con el entorno en lugar del operario para realizar

una seria de tareas concretas. Normalmente existe un lazo interno de realimentación que hace que los robots se comporten de manera lineal y permite conseguir el mayor grado posible de telepresencia, es decir, que permita al operador realizar tareas con tanta destreza como si manipulara directamente el entorno.

Un sistema de teleoperación consta de los siguientes elementos:

- Operador o teleoperador: es un ser humano que realiza el control a distancia. Su acción puede ir desde un control continuo hasta una intervención intermitente, con la que únicamente se ocupa de monitorizar y de indicar objetivos y planes cada cierto tiempo.
- Dispositivo teleoperado: es la máquina que trabaja en la zona remota y que está siendo controlada por el operador.
- Interfaz: conjunto de dispositivos que permiten la interacción del operador con el sistema de teleoperación, como por ejemplo monitores de video o el propio manipulador maestro. En definitiva, cualquier dispositivo que permita al operador mandar información al sistema y recibirla.
- Control y canales comunicación: conjunto de dispositivos que modulan, transmiten y adaptan el conjunto de señales que se transmiten entre la zona remota y la local. Generalmente se contará con uno o varias unidades de procesamiento.
- Sensores: conjunto de dispositivos que recogen la información, tanto de la zona local como de la zona remota, para ser utilizada por el interfaz y el control.

2.3. Arquitectura de un sistema teleoperado

Las diversas arquitecturas de control en teleoperación se diferencian básicamente por la información que se intercambia entre el maestro y el esclavo y por el tipo de sensores que se requieren. Según estos parámetros podemos distinguir las siguientes categorías:

- Esquema Posición-Posición: la posición del esclavo se determina a partir de la del maestro y viceversa. No hay necesidad de sensores de fuerza.
- Esquema Fuerza-Posición: la posición del esclavo se determina por seguimiento del robot maestro y las fuerzas que aparecen sobre el esclavo se miden y se generan en el maestro mediante sus motores. Sólo se requiere medida de fuerzas en el esclavo.

- Esquema Fuerza-Fuerza: las trayectorias del maestro y el esclavo se determinan a partir de las lecturas de fuerza de ambos. También existe un control local de posición en ambos robots.
- Esquema de Cuatro Canales: en este caso hay intercambio de información tanto en posición como en fuerza. El análisis teórico de esta solución refleja que es capaz de proporciona transparencia infinita.

Estas arquitecturas permiten diferentes grados de telepresencia al operador la cual se ve afectada principalmente por la estabilidad y y la transparencia en el sistema.

La estabilidad es condición indispensable en un sistema teleoperado ya que no es admisible que el robot empiece a sacudirse mientras el operador está manejándolo. Tanto el ruido eléctrico como los retrasos en las señales afectan a la estabilidad pero, mientras que el ruido es un síntoma común en cualquier sistema real, los retrasos sólo se dan cuando el robot se encuentra muy alejado.

Por otro lado el concepto de transparencia está mucho más ligado a la teleoperación. Para que las tareas realizadas por el esclavo sean lo suficientemente precisas no vale únicamente con el control de la posición del esclavo, sino que también es necesario percibir las fuerzas que aparecen sobre el esclavo, haciendo referencia a lo que se conoce como interfaces hápticas.

2.4. Aplicaciones

Los robots teleoperados pueden encontrarse en la industria nuclear (mantenimiento de reactores), química (manejo a distancia de sustancias peligrosas o tóxicas), militar (detección, manipulación y desmantelamiento de cargas explosivas), desminado humanitario, espacial (exploraciones realizadas en la luna y en marte, también en transbordadores espaciales), minera (excavaciones, manejo de cargas explosivas en minas y túneles), en el sector de seguridad, mantenimiento y rescate (inspección de sistemas de alcantarillado y tuberías, reconocimiento de zonas de desastres), telecirugía , entre muchas otras áreas.

3. Herramientas utilizadas

En este apartado describiremos los elementos físicos utilizados para la realización del proyecto. Principalmente se trata del dispositivo Kinect y los elementos que lo componen y del robot TEO.

3.1. Kinect

Antes de hablar del dispositivo Kinect se introducirá el tema de la visión artificial con el fin de conocer un poco como surgieron y en qué se basan los dispositivos de visión artificial.

3.1.1. Visión artificial

La visión artificial o visión por computadora consiste en la captación de imágenes mediante cámaras y su posterior tratamiento mediante técnicas de procesamiento avanzadas realizadas por una computadora.

Historia

El inicio de la visión artificial, desde el punto de vista práctico, fue marcado por Larry Roberts, el cual, en 1961 creó un programa que podía "ver" una estructura de bloques, analizar su contenido y reproducirla desde otra perspectiva, demostrando así a los espectadores que esa información visual que había sido mandada al ordenador por una cámara, había sido procesada adecuadamente por él.

A pesar de que se depositaron muchas esperanzas en esta nueva tecnología, hubo que esperar a los 90 para que los sensores y ordenadores aumentasen sus prestaciones y la visión por computadora se extendiese a muchas otras aplicaciones. Uno de los sectores donde más se extendió fue la industria, concretamente en las plantas de producción y cadenas de ensamblado, donde las cámaras se usaban para inspeccionar los productos fabricados y servían para obtener información para los robots guiados.

Con el paso del tiempo se han mejorado tanto las cámaras como los ordenadores, gracias a lo cual la visión artificial se ha extendido a muchas otras aplicaciones como son:

- La detección, segmentación, localización y reconocimiento de ciertos objetos en imágenes (por ejemplo, caras humanas).
- La evaluación de los resultados (por ejemplo, segmentación, registro).
- Registro de diferentes imágenes de una misma escena u objeto, es decir, hacer concordar un mismo objeto en diversas imágenes.
- Seguimiento de un objeto en una secuencia de imágenes.
- Mapeo de una escena para generar un modelo tridimensional de la escena; este modelo podría ser usado por un robot para navegar por la escena.
- Estimación de las posturas tridimensionales de humanos.
- Búsqueda de imágenes digitales por su contenido.

Bases científicas

En el caso de la visión humana es el ojo el encargado de transformar la energía luminosa en señales electroquímicas que se envían al cerebro para su posterior procesamiento.

Sin embargo, en la visión artificial se sustituye el ojo por una cámara de vídeo (o cualquier dispositivo similar), que se encarga de convertir la energía luminosa en corriente eléctrica (cámaras analógicas) o señales digitales (cámaras digitales).

Para poder capturar las imágenes las cámaras poseen matrices de receptores. Cada elemento de la matriz se le conoce como píxel y el valor que toma al incidir sobre el la luz se llama “intensidad” o “nivel de gris”. Los elementos fotosensivos pueden ser de diversas tecnologías como CCD o CMOS.

En el caso de querer obtener una imagen en color deberemos utilizar una cámara RGB, la cual tendrá 4 receptores por pixel que se encargan de obtener la intensidad de rojo, azul y verde como se muestra en la figura 1. Aunque generalmente de cada mosaico repetitivo de 2x2 dos de sus filtros serán del mismo tono de verde, se pueden tomar verdes de diferentes tonos para obtener una imagen más fiel a cambio de coste computacional. Tras la captación de la luz se llevará a cabo una interpolación para crear la imagen final.

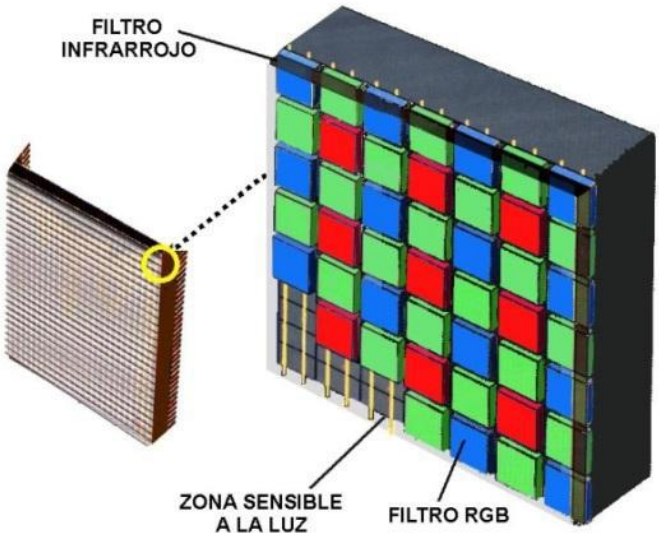

Figura 1. Sensor CCD formato RGB

3.1.4. Dispositivo Kinect

Figura 2. Dispositivo Kinect

El dispositivo Kinect, denominado originalmente como *Proyect Natal*, nació como un nuevo controlador para la videoconsola Xbox 360 y desde Junio de 2011 para PC a través de Windows 7 y Windows 8. Creado por Alex Kipman y desarrollado por Microsoft, su principal característica consiste en que permite a los usuarios controlar e interactuar con la consola sin necesidad de tener contacto físico con un controlador de videojuegos tradicional, mediante una interfaz natural de usuario que reconoce gestos, comandos de voz, y objetos e imágenes.

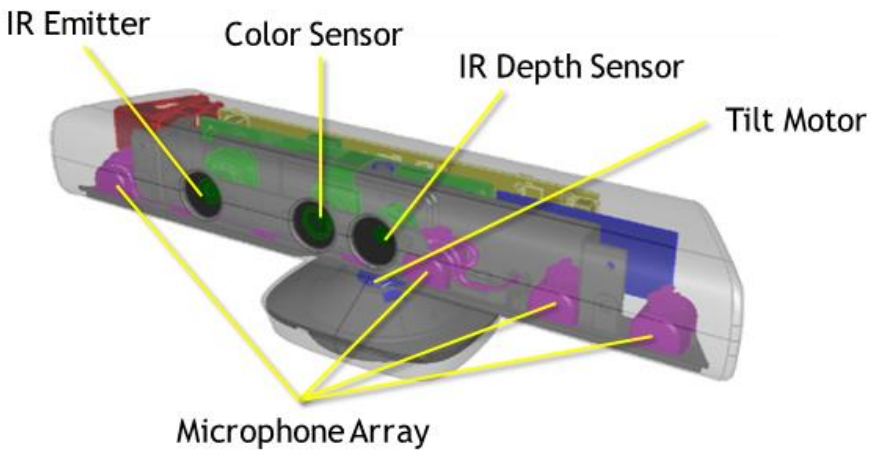

Figura 3. Elementos principales del dispositivo Kinect

La primera versión se dio a conocer por primera vez en el E3 (Electronic Enterteiment Expo) de 2009 con el nombre anteriormente mencionado de *Proyect Natal*, pero no fue hasta un año después en el E3 de 2010 que se dio a conocer su nombre definitivo: Kinect.

Dicho dispositivo cuenta con una cámara RGB y otra NIR, un sensor de profundidad, una fuente de luz infrarroja que permite a la cámara NIR operar en la oscuridad, un micrófono *multi-array* y un procesador personalizado que ejecuta el software patentado, que proporciona captura de movimiento de todo el cuerpo en 3D, reconocimiento facial y capacidades de reconocimiento de voz. El micrófono de Kinect permite a la Xbox 360 llevar a cabo la localización de la fuente acústica y la supresión de ruido ambiental. Además cuenta con un motor que permite variar la inclinación de la Kinect y unos acelerómetros para medir con mayor precisión la posición de la cámara. A continuación pasaremos a describir los elementos más importantes para el proyecto: el sensor de profundidad y la cámara RGB.

Sensor de profundidad

El sensor de profundidad está formado por un emisor de laser infrarrojos, un sensor CMOS monocromo y el módulo encargado de construir la nube de puntos tridimensional. La medición se realizará a una velocidad de 30 fps y tomando *frames* con una resolución de 320x240 y 16 bits por píxel.

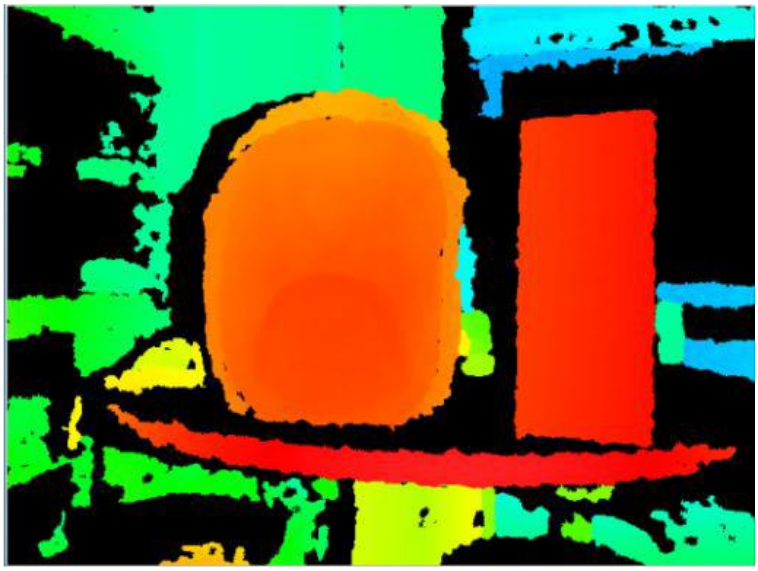

Figura 4. Imagen de la cámara de profundidad

Para calcular la profundidad Kinect se utiliza un emisor infrarrojo que genera un haz láser que proyecta un patrón de puntos sobre los cuerpos pero, a diferencia de los sistemas infrarrojos más caros basados en “tiempo de vuelo” que calculan la profundidad midiendo el tiempo desde que se lanzó el pulso láser hasta que se recibió, esta cámara infrarroja capta este patrón y el software calcula la profundidad de cada punto midiendo la disparidad entre píxeles y la deformación de la malla emitida triangulando las distancias entre puntos detectados y comparándolos con su localización original. El patrón de luz emitida por la cámara se puede observar en las imágenes cuando no se muestran a máxima resolución.

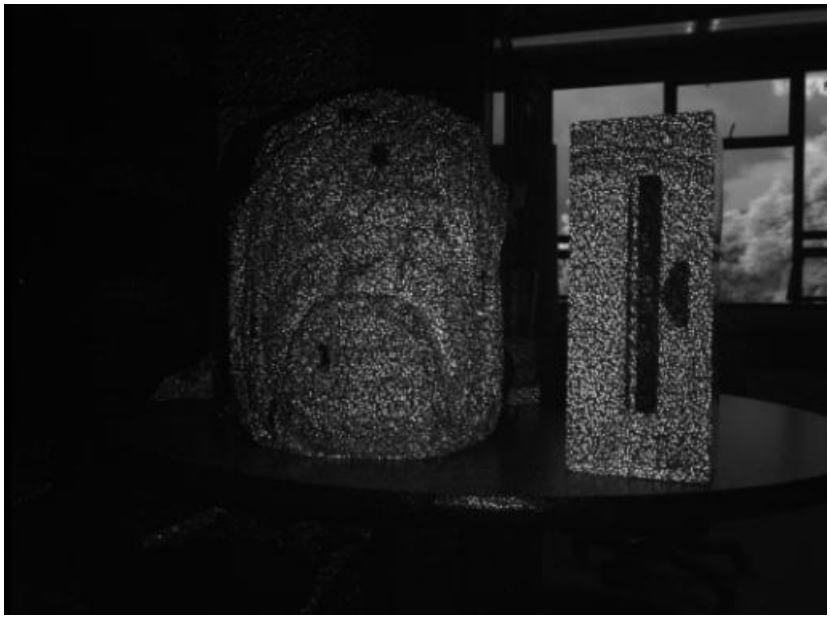

Figura 5. Malla de puntos infrarrojos

El tamaño de los puntos del patrón infrarrojo nos proporciona información de la distancia al sensor. Se pueden distinguir varias zonas de funcionamiento, en las que la precisión del sensor variará según la zona:

- Desde los 0.8 m hasta los 1.2 m la precisión de las medidas es muy alta debido a la gran cantidad de puntos que contendrán los objetos, por lo que la precisión a la hora de conseguir la forma de los objetos será alta.
- Desde 1.2 m hasta los 2 m las medidas tomadas poseen algunos puntos menos pero la precisión en las medidas siguen siendo buenas.
- A partir de los 2m la perdida de información comienza a ser significativa, pudiendo detectar las siluetas pero no algunas formas de los objetos. Cuando llegamos a los 8m la perdida de información es lo suficientemente grande como para desestimar la detección de personas.

Cámara RGB

La denominada por Microsoft como cámara RGB, se trata de una cámara VGA a color fabricada con un sensor CMOS, con una resolución de 640x480 pixeles, una frecuencia de muestreo de 30 fps y un tamaño de 32 bits por pixel. Su funcionamiento es óptimo entre los 40cm y los 4m, aunque da resultados válidos hasta los 8m.

Su rango de visión es de 57º en el plano horizontal y de 43º en el vertical, a los que se suman 27º debido a la posibilidad de inclinar la cámara gracias al motor que incorpora en la base de apoyo.

Montaje del dispositivo Kinect

Figura 6. Despiece del dispositivo Kinect

Incluyo este apartado ya que cuando se me entregó el dispositivo Kinect que tendría que utilizar estaba completamente desmontado debido a que había sido necesario desmontarlo en un proyecto anterior. Para realizar el ensamblado y conexionado de todas las partes primero fue necesario investigar sobre la estructura interna de Kinect, en qué posición iba cada pieza y cómo iban conectados los distintos cables. Fue bastante complicado encontrar información sobre estas cuestiones ya que prácticamente ninguna de las páginas que contenían algún tipo de información sobre Kinect versaban sobre lo que a mí me interesaba. Sin embargo, encontré un par de videos en los que desmontaban por completo el dispositivo y gracias a que tenían una definición bastante decente pude hacerme una idea de cómo tenía que montar la Kinect.

Cuando por fin termine de montar las piezas y conectar los cables me dispuse a conectarlo al ordenador para probarlo y, sorpresa, el ordenador no detectaba el dispositivo. Tras revisar por completo el dispositivo varias veces llegué a la conclusión de que el problema podía estar en la conexión de seis pines de la placa principal a la cual iba conectado el cable que, a su vez, estaba formado por el cable de alimentación de 12 V conectado al enchufe del

dispositivo, el cable de alimentación de 5V conectado al USB del dispositivo, los cables de datos Data+ y Data- conectados también al USB del dispositivo y, por último dos cables de tierra. Dicha conexión estaba un poco defectuosa por lo que se decidió comprobar si el dispositivo estaba correctamente alimentado. Tras comprobar la continuidad de cada uno de los cables se vio que, a pesar de que los cables no estaban dañados, la alimentación del dispositivo no era correcta, lo que probablemente se debía a que algunos pines no estaban bien fijados a los cables. De esta forma se decidió soldar los pines a los cables para asegurar el contacto entre ambos y, tras volver a conectar el dispositivo al ordenador seguía sin funcionar y, aunque esta vez la alimentación del dispositivo si era la correcta, el dispositivo no recibía ningún tipo de información por los pines Data+ y Data-. En este punto se decidió cambiar la placa principal de la Kinect por otra extraída de otra Kinect que se sabía que funcionaba correctamente. Al hacerlo y conectar la Kinect al ordenador este la reconoció inmediatamente por lo que se llegó a la conclusión de que el problema no era de los cables sino se algún elemento de la placa principal de la Kinect.

Dado que encontrar y solucionar el problema podía ser muy tedioso y llevar demasiado tiempo debido a que los componentes de la placa no eran de uso comercial sino que específicos para el dispositivo, se decidió usar una Kinect distinta para el proyecto.

3.2. Robot TEO

Aunque el nombre actual del robot humanoide utilizado en el proyecto es TEO se trata de una evolución del robot humanoide RH-2 utilizado por la universidad en anteriores proyectos. Se trata de un modelo perfeccionado de sus predecesores en la Universidad Carlos III (RH-0 y RH-1) de 1,65m de altura, 60 kg de peso y hasta 26 grados de libertad.

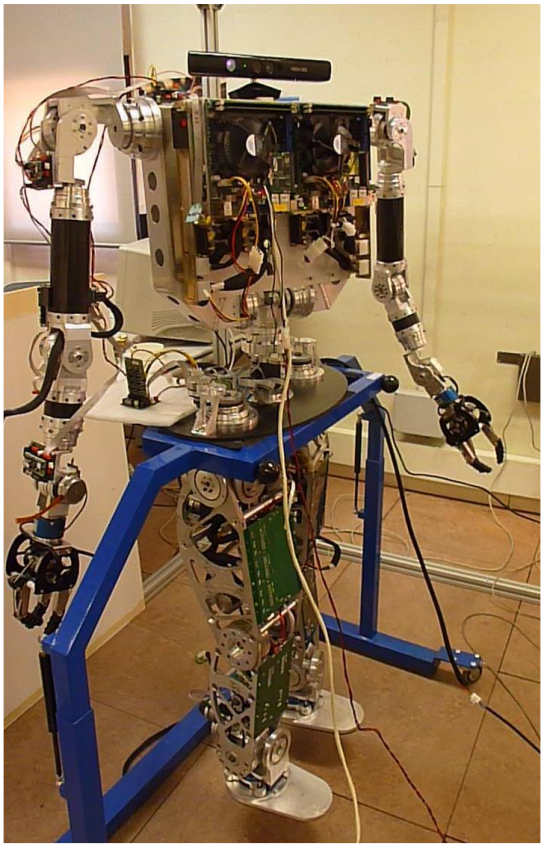

Figura 7. TEO

Concretamente los GDL de los brazos se distribuyen de la siguiente manera para imitar la capacidad del movimiento humano:

- 3 GDL en el hombro correspondiéndose con los planos sagital, frontal y transversal.
- 1 GDL en el codo en el plano sagital.
- 2 GDL en la muñeca coincidiendo con los planos transversal y frontal.

3.3. Visual Studio 2010 Express

Como entorno de programación se ha elegido Visual Studio 2010 Express ya que se puede descargar de forma gratuita desde la web de Microsoft y cubre nuestras necesidades respecto al proyecto.

Microsoft Visual Studio es un entorno de desarrollo integrado (IDE, por sus siglas en inglés) para sistemas operativos Windows. Soporta múltiples lenguajes de programación tales como C++, C#, Visual Basic .NET, F#, Java, Python, Ruby, PHP. A partir de la versión 2005 Microsoft ofrece gratuitamente las *Ediciones Express*, que son versiones básicas separadas por lenguajes de programación o plataforma enfocadas; para estudiantes y programación amateur. Estas ediciones son iguales al entorno de desarrollo comercial, pero sin características avanzadas de integración. En la versión de 2010 podemos encontrar:

- Visual Web Developer 2010 Express
- Visual Studio 2010 Express para Windows Phone
- Visual Basic 2010 Express
- Visual C# 2010 Express
- Visual C++ 2010 Express

Dado que a programación se iba a realizar en C++ se escogió la versión Visual C++ 2010 Express. Podremos descargar el programa y encontrar información adicional en el siguiente enlace: http://www.microsoft.com/visualstudio/esn/downloads#d-2010-express

3.4. Kinect for Windows SDK

Kinect for Windows SDK es el software oficial de Windows para desarrolladores para trabajar con Kinect en un PC. La primera versión (beta) apareció en Junio de 2011 de forma gratuita y estaba pensada para el desarrollo de aplicaciones en Windows 7. Incluía varios ejemplos que ayudaban a comprender el funcionamiento del software y permitía programar en C++, C# y Visual Basic.

Finalmente, y tras el lanzamiento de una segunda versión beta, en Febrero de 2012 sacaron la primera versión final (1.0), la cual no solo había sufrido varias mejoras respecto a las versiones anteriores sino que, además, permitía comercializar aquellas aplicaciones que desarrollásemos para Windows.

La versión seleccionada para el proyecto es la v.1.8, la última versión disponible cuando se empezó el proyecto, aunque posteriormente ha salido una nueva versión v.2.0. Podremos adquirir Kinect for Windows y obtener una guía de uso en la dirección http://www.microsoft.com/en-us/kinectforwindowsdev/Downloads.aspx

De los ejemplos facilitados por el programa el más útil a la hora de llevar a cabo el proyecto es *SkeletalViewer.* Dicho ejemplo demuestra como el dispositivo Kinect toma imágenes a través del sensor de profundidad, la cámara RGB y realiza la esqueletización mediante 20 ejes o puntos de una o varias personas, mostrando todo esto por pantalla a través de la ventana principal de la aplicación. Este ejemplo era perfecto ya que parte del proceso del consiste en calcular la posición en el espacio de cada uno de los 20 puntos o ejes detectados, con lo que se podían extraer dichos datos y realizar los cálculos necesarios con ellos. Para comprender el funcionamiento interno del programa descargar el documento Skeletal Viewer Walkthrough (link para descargarlo http://es.scribd.com/doc/94544035/SkeletalViewer-Walkthrough).

3.5. OpenRAVE

Open Robot Automation Virtual Environement u OpenRAVE provee de un entorno para probar y desarrollar algoritmos usados en aplicaciones robóticas del mundo real. Se centra principalmente en la simulación y el análisis da información cinemática y geométrica relacionada con la planificación de movimiento. Básicamente trata de aumentar la fiabilidad de los sistemas de planificación de movimiento con el fin de facilitar su posterior integración en sistemas reales. Podemos descargar el programa del siguiente enlace: http://roboticslab.sourceforge.net/asibot/install_openrave_on_ubuntu.html.

3.6. CMake

CMake es una herramienta multiplataforma de generación o automatización de código separada y de más alto nivel que el sistema *make* común de Unix. CMake se utiliza para controlar el proceso de compilación del software usando ficheros de configuración sencillos e independientes de la plataforma, es decir, genera *makefiles* nativos y espacios de trabajo que pueden usarse en el entorno de desarrollo deseado mediante un proceso controlado por ficheros de configuración llamados CMakeLists.txt. El programa soporta la generación de ficheros para varios sistemas operativos, lo que

facilita el mantenimiento y elimina la necesidad de tener varios conjuntos de ficheros para cada plataforma. Algunas de sus principales funcionalidades son:

- Ficheros de configuración escritos en un lenguaje de scripting específico para CMake.
- Análisis automático de dependencias para C, C++, Fortran, y Java.
- Soporte para varias versiones de Microsoft Visual Studio, incluyendo la 6, 7, 7.1, 8.0, 9.0 y 10.0.
- Soporte para *builds* paralelos.
- Compilador cruzado.
- Soporte para *builds* paralelos: Linux, Mac OS X, Windows 95/98/NT/2000/XP/Vista/7.

En nuestro caso utilizaremos esta herramienta en el ámbito de Linux para generar un programa que se encargue de recibir leer los datos procedentes del puerto yarp y pasarlos al simulador o al robot humanoide. Podremos obtener el programa del siguiente enlace: http://roboticslab.sourceforge.net/asibot/install_cmake_on_ubuntu.html.

3.7. YARP

YARP o Yet Another Robot Platform es la herramienta que se ha seleccionado para comunicar en tiempo real nuestro ordenador con el robot humanoide o con otros ordenadores.

YARP es un conjunto de protocolos, librerías y herramientas para mantener módulos desacoplados. Está escrito casi íntegramente en C++ y se encarga de establecer un sistema de control y transmisión de datos entre dispositivos separados. Está pensado para facilitar la evolución y desarrollo del software y los sensores en robótica, ya que las dificultades que acarrea el cambiar sensores o procedimientos siempre ha sido uno de los mayores problemas de este campo.

En el siguiente enlace podremos descargar el programa para el sistema operativo Windows: http://sourceforge.net/projects/yarp0/files/yarp2/. De querer utilizar el programa en Linux, seguiremos los pasos indicados en el siguiente enlace; http://roboticslab.sourceforge.net/asibot/install_yarp_on_ubuntu.html.En case de querer profundizar más en el uso de YARP podemos acceder a información y tutoriales en el link http://eris.liralab.it/yarpdoc/index.html.

Universidad Carlos III de Madrid

Antes de poder empezar a programar y utilizar las bibliotecas de Kinect SDK y YARP deberemos incluirlas, para ello modificaremos los siguientes puntos:

En Propiedades/Directorios de VC++/Directorios de archivo de inclusión, añadiremos las direcciones de las carpetas “include” del dispositivo Kinect y del programa YARP:

- D:\robotology\yarp-2.3.63\include
- $(KINECTSDK10_DIR)\inc)

En Propiedades/Directorios de VC++/Directorios de archivo de bibliotecas, añadiremos las direcciones de las carpetas “lib” del dispositivo Kinect y del programa YARP:

- $(KINECTSDK10_DIR)\lib\x86
- D:\robotology\yarp-2.3.63\lib

En Propiedades/Vinculador/Entrada/Dependencias adicionales, incluiremos algunas de las librerías .lib necesarias tales como:

- Kinect10.lib
- ACE.lib
- YARP_OS.lib
- YARP_init.lib

Por último, al inicio del código deberemos hacer los llamamientos a las librerías necesarias:

- #include <NuiApi.h>
- #include <yarp/os/all.h>
- #include <yarp/os/network.h>
- #include <yarp/os/port.h>
- #include <yarp/os/bottle.h>
- #include <yarp/os/time.h>
- #include <yarp/os/all.h>

Universidad Carlos III de Madrid

4. Diseño y programación

Este apartado se dedicará a explicar el proceso que se ha seguido hasta completar el código apropiado para los objetivos que se habían establecido. En nuestro caso partiremos ya del programa-ejemplo SkeletalViewer facilitado por Kinect for Windows.

Este programa utilizará la NUI API para capturar los datos de profundidad, el color y hacer el seguimiento del esqueleto. Para sacara la imágenes por pantalla se valdrá de Direct3D 9 y de Windows GDI. En la figura 8 se expone un diagrama de bloques esquemático sobre el funcionamiento general y más adelante desarrollaremos los puntos por los que pasa.

Los bloques desarrollados en el ordenador conectado a la Kinect serán de color azul mientras que los del ordenador de la simulación serán rojos.

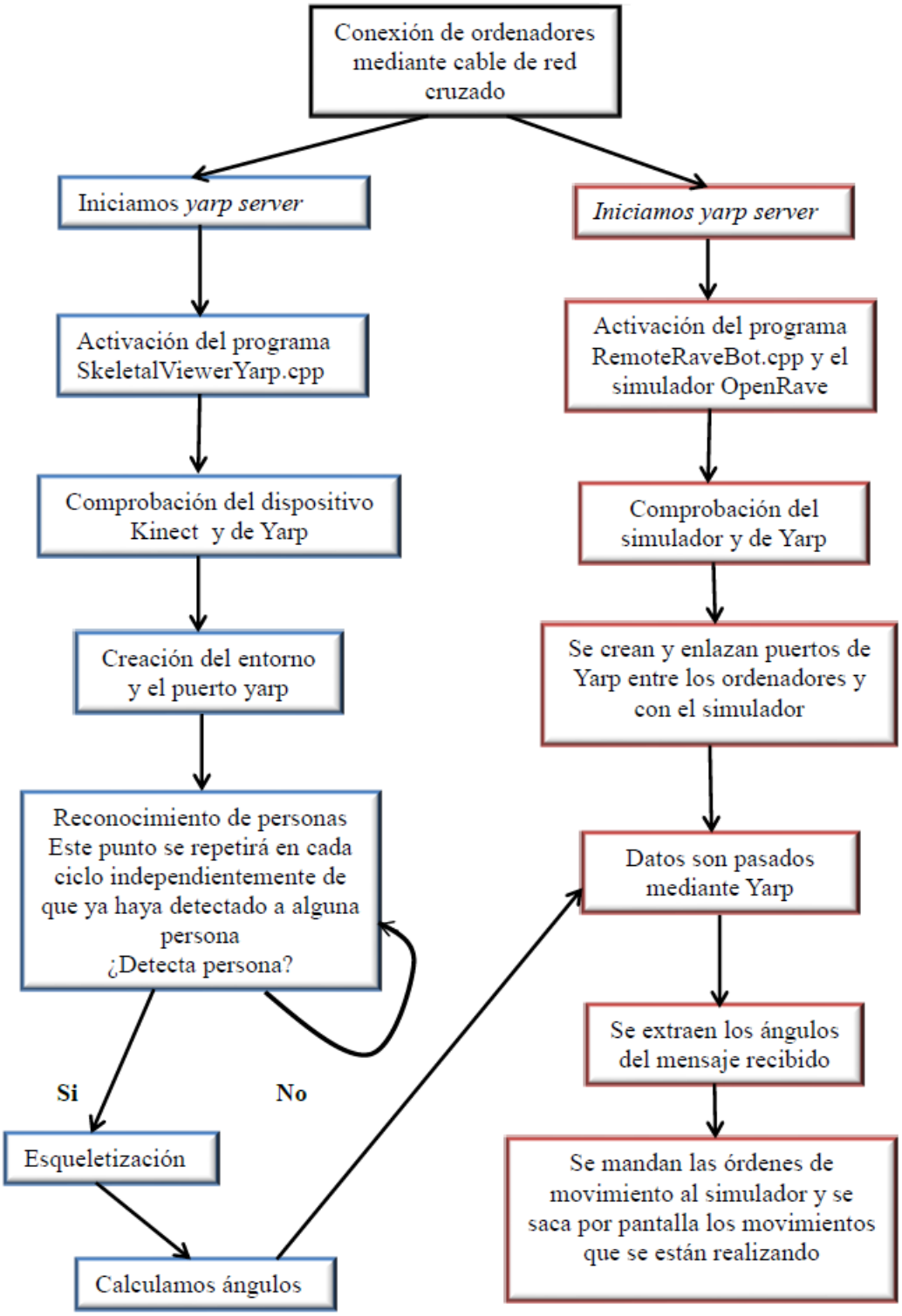

Figura 8. Diagrama de bloques

4.1. Creación de la ventana principal

El archivo SkeletalViewer.cpp define el punto de entrada de la aplicación y su función de llamada al procesamiento de la ventana. Cuando se crea un nuevo proyecto, el programa genera el archivo fuente *project.cpp*, el cual incluye un punto de entrada llamado *_tWinMain* y una función de llamada al procesamiento de la ventana llamada *WndProc.*

La función *_tWinMain* crea y muestra por pantalla la ventana principal de la aplicación de la siguiente manera:

```
//-------------------------------------------------------------------
// _tWinMain
//
// Entry point for the application
//-------------------------------------------------------------------
int  APIENTRY  _tWinMain(HINSTANCE  hInstance,  HINSTANCE  hPrevInstance,  LPTSTR
lpCmdLine, int nCmdShow)
{
    MSG       msg;
    WNDCLASS  wc;

    // unique mutex, if it already exists there is already an instance of this
    app running
    // in that case we want to show the user an error dialog
    HANDLE hMutex = CreateMutex( NULL, FALSE, INSTANCE_MUTEX_NAME );
    if ( (hMutex != NULL) && (GetLastError() == ERROR_ALREADY_EXISTS) )
    {
        TCHAR szAppTitle[256] = { 0 };
        TCHAR szRes[512] = { 0 };

        //load the app title
        LoadString( hInstance, IDS_APPTITLE, szAppTitle, _countof(szAppTitle) );

        //load the error string
        LoadString( hInstance, IDS_ERROR_APP_INSTANCE, szRes, _countof(szRes) );

        MessageBox( NULL, szRes, szAppTitle, MB_OK | MB_ICONHAND );

        CloseHandle(hMutex);
        return -1;
    }
    // Store the instance handle
    g_skeletalViewerApp.m_hInstance = hInstance;

    // Dialog custom window class
    ZeroMemory( &wc,sizeof(wc) );
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.cbWndExtra = DLGWINDOWEXTRA;
    wc.hInstance = hInstance;
    wc.hCursor = LoadCursor(NULL,IDC_ARROW);
```

```
    wc.hIcon = LoadIcon(hInstance,MAKEINTRESOURCE(IDI_SKELETALVIEWER));
    wc.lpfnWndProc = DefDlgProc;
    wc.lpszClassName = SZ_APPDLG_WINDOW_CLASS;
    if( !RegisterClass(&wc) )
    {
        return 0;
    }

    // Create main application window
    HWND hWndApp = CreateDialogParam(
        hInstance,
        MAKEINTRESOURCE(IDD_APP),
        NULL,
        (DLGPROC) CSkeletalViewerApp::MessageRouter,
        reinterpret_cast<LPARAM>(&g_skeletalViewerApp));
    // Show window
    ShowWindow(hWndApp,nCmdShow);
    // Main message loop:
    while( GetMessage( &msg, NULL, 0, 0 ) )
    {
        // If a dialog message will be taken care of by the dialog proc
        if ( (hWndApp != NULL) && IsDialogMessage(hWndApp, &msg) )
        {
            continue;
        }

        // otherwise do our window processing
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    CloseHandle( hMutex );

    return static_cast<int>(msg.wParam);
}
```

Figura 9. Creación de la ventana principal

La ventana consiste en una simple caja de diálogo que no admite entradas provenientes del usuario, por lo que dicha ventana no se puede modificar.

4.2. Inicialización de la NUI API

Una vez creada la ventana principal de la aplicación se ejecuta la función *g_SkeletalViewerApp::Nui_Init*, la cual realiza la mayor parte del trabajo requerido para inicializar el sensor Kinect y comenzar la transmisión de datos de la siguiente forma:

- Crea un evento que es asociado con cada tipo de dato. El programa espera a que un evento determine que acción realizar.
- Inicializa estructuras para el renderizado del esqueleto.
- Inicializa estructuras para el renderizado de imagen.
- Inicializa el sensor Kinect.
- Abre canales de transmisión para las imágenes de profundidad y color.
- Crea el hilo de procesamiento de datos del sensor Kinect.

Cuando una imagen esta lista para la aplicación, se crea un evento para cada uno de los tres tipos de datos llamando a la función *CreateEvent* como se indica a continuación:

```
    m_hNextDepthFrameEvent  =  CreateEvent(  NULL,  TRUE,  FALSE,
NULL );
    m_hNextColorFrameEvent  =  CreateEvent(  NULL,  TRUE,  FALSE,
NULL );
    m_hNextSkeletonEvent = CreateEvent( NULL, TRUE, FALSE, NULL
);
```

Figura 10. Creación del evento para uno de los tres tipos de datos

Posteriormente *Nui_Init* crea la estructura y el mapa de bits que requiere el programa para renderizar cada tipo de dato proveniente del sensor.

4.3. Inicialización del renderizado del esqueleto

El programa dibuja el esqueleto basándose en el subsistema de Windows GDI, el cual proporciona una salida independiente del dispositivo. La función *CSkeletalViewerApp::Nui_Init* ajusta las estructuras GDI requeridas de la siguiente forma:

```
    GetWindowRect(  GetDlgItem(  m_hWnd,  IDC_SKELETALVIEW  ),  &rc
);
    HDC hdc = GetDC( GetDlgItem( m_hWnd, IDC_SKELETALVIEW) );

    int width = rc.right - rc.left;
    int height = rc.bottom - rc.top;
```

```
    m_SkeletonBMP = CreateCompatibleBitmap( hdc, width, height
);
    m_SkeletonDC = CreateCompatibleDC( hdc );

    ReleaseDC(GetDlgItem(m_hWnd,IDC_SKELETALVIEW), hdc );
    m_SkeletonOldObj = SelectObject( m_SkeletonDC, m_SkeletonBMP
);
```

Figura 11. Ajuste de las estructuras GDI

GDI maneja todas las operaciones específicas del dispositivo en nombre de la aplicación. Cuando el mapa de bits está listo para mostrar por pantalla, el programa llama a GDI para mostrarlo.

4.4. Inicialización del renderizado de la imagen

El programa usa Direct3D 9 para renderizar imágenes de video y profundidad, por lo que cuando es necesario la aplicación llama a Direct3d 9 para mostrar las imágenes en secuencia.

La clase *DrawDevice* contiene métodos para crear y manipular estructuras de Direct3D 9. El programa declara *m_DrawDepth* y *m_DrawVideo*, los cuales representan objetos de *DrawDevice* para las imágenes de video y profundidad. *CSkeletalViewerApp::Nui_Init* crea e inicializa los dispositivos Direct3D 9 y los mapas de bits que la aplicación usa con cada uno de esos objetos como sigue:

```
hr      =      m_DrawDepth.CreateDevice(      GetDlgItem(m_hWnd,
IDC_DEPTHVIEWER ) );
hr = m_DrawDepth.SetVideoType( 320, 240, 320 * 4 );
```

Figura 11. Creación e inicialización de los dispositivos Direct3D9

4.5. Inicialización del sensor Kinect

NuiInitialize inicializa el motor interno de captura de imágenes, el cual inicia un canal que recupera los datos del sensor Kinect e indica a la aplicación cuando una imagen está lista.

Después de que *Nui_Init* configure las estructuras requeridas para el renderizado, inicializa el sensor Kinect de la siguiente forma:

```
hr = NuiInitialize(

      NUI_INITIALIZE_FLAG_USES_DEPTH_AND_PLAYER_INDEX |

      NUI_INITIALIZE_FLAG_USES_SKELETON |

      NUI_INITIALIZE_FLAG_USES_COLOR);
```

Figura 13. Inicialización de la captura de imágenes

4.6. Procesamiento de los datos del sensor

El procesamiento de daros del sensor en el programa está controlado por un bucle en el método *CSkeletalViewerApp::Nui_ProcessThread.* El siguiente código muestra el flujo básico del bucle:

```
while(1)
{
     // Wait for event.
     // If the stop event occurs, stop looping and exit
     // Calculate frames per second.
     // Blank the skeleton display.

     // Process frame events
     switch(EventType) {
          case Depth:
               Nui_GotDepthAlert();
               break;
          case Video:
               Nui_GotVideoAlert();
               break;
          case Skeleton:
               Nui_GotSkeletonAlert( );
               break;
     }
}
```

Figura 12. Bucle principal de procesamiento

El bucle espera a que ocurra un evento. Si ocurre un evento de parada, el bucle deja de procesar y sale del canal de procesamiento. SI el evento indica que una imagen de profundidad, de esqueleto o de color está lista, el bucle realiza los siguientes pasos:

Universidad Carlos III de Madrid

- Calcula el número de imágenes por segundo y muestra este número por pantalla.
- Toma y bloquea una parte de la imagen en la que se considera que se encuentran los datos del esqueleto.
- Procesa la imagen.

4.7. Esqueletización

Este programa se encarga de asegurarse que el esqueleto rastreado no tiene partes fuera de la pantalla, es capaz de rastrear de forma activa a 2 esqueletos a la vez diferenciándolos y de forma pasiva a otros 4. Cada esqueleto será representado por un color diferente y la anchura del trazo vendrá dada por la anchura del espacio destinado a ese esqueleto dividida por 80 como se muestra en el siguiente bloque de código:

```
m_Pen[0] = CreatePen( PS_SOLID, width / 80, RGB(0, 255, 0) );
m_Pen[1] = CreatePen( PS_SOLID, width / 80, RGB( 0, 0, 255 ) );
m_Pen[2] = CreatePen( PS_SOLID, width / 80, RGB( 255, 0, 0 ) );
m_Pen[3] = CreatePen( PS_SOLID, width / 80, RGB(255, 255, 64 )
);
m_Pen[4] = CreatePen( PS_SOLID, width / 80, RGB( 255, 64, 255 )
);
m_Pen[5] = CreatePen( PS_SOLID, width / 80, RGB( 128, 128, 255 )
);
```

Figura 13. Establecer color y grosor

En un primer momento el esqueleto es dispuesto en función de las coordenadas del plano de profundidad ya que desde ahí se han sacado los datos. Más tarde se deberá pasar al sistema final, (160, 120) en coordenadas de la imagen de profundidad.

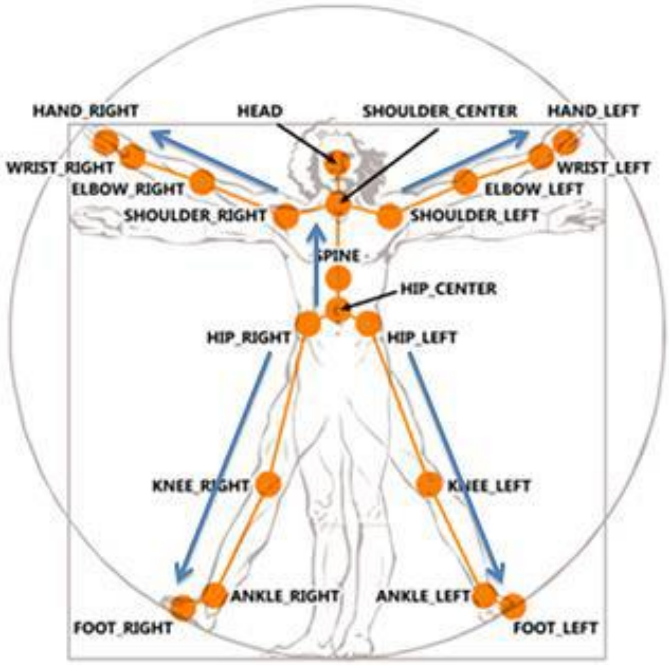

Figura 14. Puntos característicos

Los datos que se han obtenido son los puntos que representan partes representativas del cuerpo como centro de la cabeza, pecho o la cadera tal como aparecen en la figura 14. Se dibujarán segmentos entre los puntos calculados utilizando la función “Polyline” como aparece en la figura 15 y más tarde cada punto característico será resaltado en un color como se aprecia en la figura 16.

```
void                CSkeletalViewerApp::Nui_DrawSkeletonSegment(
NUI_SKELETON_DATA * pSkel, int numJoints, ... )
{
 va_list vl;
 va_start(vl,numJoints);
 POINT segmentPositions[NUI_SKELETON_POSITION_COUNT];

 for (int iJoint = 0; iJoint < numJoints; iJoint++)
 {
    NUI_SKELETON_POSITION_INDEX          jointIndex          =
va_arg(vl,NUI_SKELETON_POSITION_INDEX);
    segmentPositions[iJoint].x = m_Points[jointIndex].x;
    segmentPositions[iJoint].y = m_Points[jointIndex].y;
}
Polyline(m_SkeletonDC, segmentPositions, numJoints);
```

Figura 15. Crea los segmentos entre ejes

```
    for ( i = 0; i < NUI_SKELETON_POSITION_COUNT ; i++ )
    {
      if     (     pSkel->eSkeletonPositionTrackingState[i]     !=
NUI_SKELETON_POSITION_NOT_TRACKED )
        {
          HPEN hJointPen;

          hJointPen      =      CreatePen(      PS_SOLID,      9,
g_JointColorTable[i]);
          hOldObj = SelectObject( m_SkeletonDC, hJointPen );

          MoveToEx(  m_SkeletonDC,  m_Points[i].x,  m_Points[i].y,
NULL);
          LineTo( m_SkeletonDC, m_Points[i].x, m_Points[i].y );

          SelectObject( m_SkeletonDC, hOldObj );
          DeleteObject( hJointPen );
        }
    }
```

Figura 16. Dibujar un punto del color estipulado en cada eje

El proceso se repetirá para todos los esqueletos detectados y una vez logrado se almacenará en la memoria compatible con la imagen y se sacará por pantalla. El resultado puede apreciarse en la figura 17.

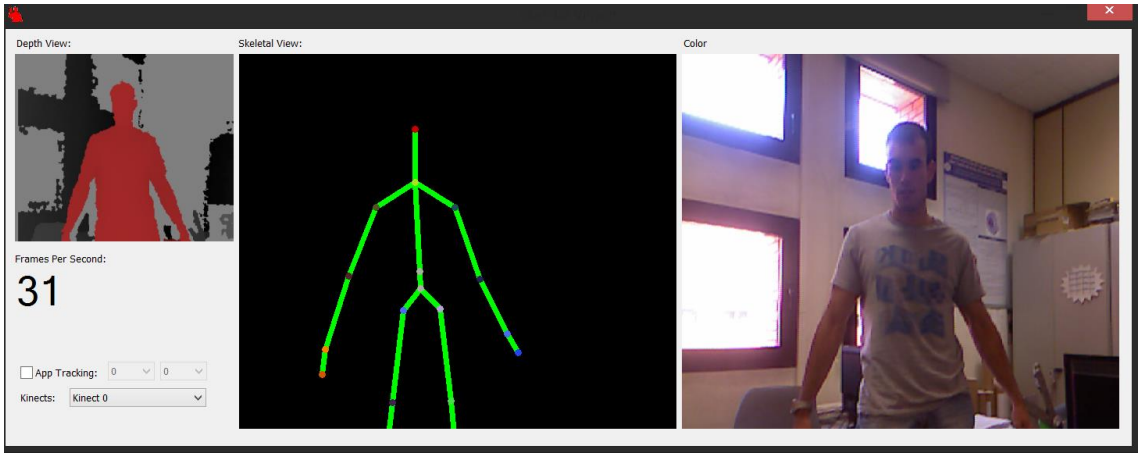

Figura 17. Esqueletización completa

4.8. Reconocimiento gestual y RPY

Una vez que el programa ha realizado la esqueletización, lo siguiente que hay que hacer es obtener los datos que nos interesan para el proyecto. Se decidió utilizar un método bastante intuitivo conocido como RPY (Roll Pitch Yaw) que, cómo se explica más adelante, una vez que se utilizó en el simulador se comprobó que no era el método más adecuado, pero se utilizó de todas formas ante la falta de tiempo para implementar otro método.

Para llevar a cabo el método RPY empezaremos trabajando con el brazo derecho, por lo que los datos que nos interesan son los ángulos que forma el hombro en los tres ejes cartesianos (*angulo_hombro_xy, angulo_hombro_xz, angulo_hombro_yz*) así como los ángulos formados por el codo (*angulo_codo_xy, angulo_codo_xz, angulo_codo_yz)* también en los tres ejes cartesianos.

El primer paso será localizar los puntos característicos que necesitaremos y aparecen resaltados en la figura 20, en nuestro caso serán la base del cuello, el hombro derecho, el codo derecho y la muñeca derecha. Usaremos los datos del espacio de coordenadas del esqueleto ya que nos ofrece las coordenadas de cada punto en los ejes X, Y y Z con centro de coordenadas en el centro del sensor de profundidad. Dado que no contamos con el efecto espejo el brazo de interés se situará a la izquierda.

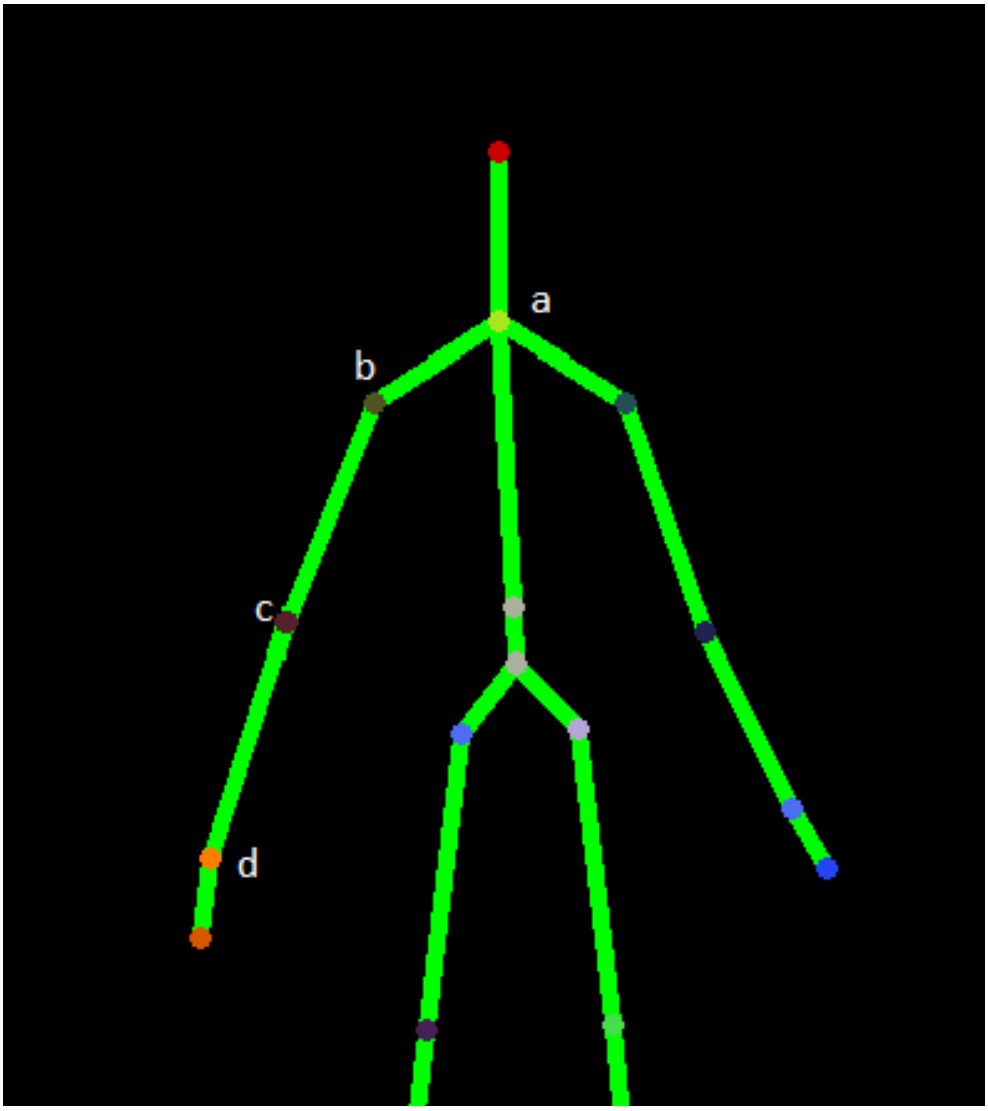

Figura 18. Base de la cinemática

Pasaremos ahora a explicar el método de cálculo utilizado para los ángulos del hombro y posteriormente los del codo. Para calcular los ángulos del hombro contaremos con tres vértices formados por base del cuello, hombro y codo, los cuales forman un triángulo del cual podremos sacar sus lados permitiéndonos así obtener los ángulos mediante el método RPY.

Lo primero que se hace es sacar los lados del triángulo abc del siguiente modo:

$$D_{xy} = \sqrt{D_x^2 + D_y^2}$$

$$A_{xy} = \sqrt{A_x^2 + A_y^2}$$

$$E_{xy} = \sqrt{E_x^2 + E_y^2}$$

Donde:

- D_x es la distancia del punto a al b en el eje X
- D_y es la distancia del punto a al b en el eje Y
- A_x es la distancia del punto b al c en el eje X
- A_y es la distancia del punto b al c en el eje Y
- E_x es la distancia del punto c al a en el eje X
- E_y es la distancia del punto c al a en el eje Y
- D_{xy} es la distancia del punto a al b en el plano XY
- A_{xy} es la distancia del punto b al c en el plano XY
- E_{xy} es la distancia del punto c al a en el plano XY

Ahora utilizamos el teorema del coseno para calcular el ángulo del hombro:

$$angulo\ hombro_{xy} = \arccos\left(\frac{A_{xy}^2 + D_{xy}^2 - E_{xy}^2}{2 * A_{xy} * D_{xy}}\right) * \frac{180}{\pi}$$

Donde:

- *angulo hombro*$_{xy}$ es el ángulo que forma la recta ab con la recta bc en el plano XY

A continuación se muestra el código que realiza dichas operaciones en los tres planos:

```
Dx=hombrox-Centrox;
Dy=hombroy-Centroy;
Dz=hombroz-Centroz;
Dxy=sqrt((Dx*Dx)+(Dy*Dy));
Dxz=sqrt((Dx*Dx)+(Dz*Dz))
Dyz=sqrt((Dy*Dy)+(Dz*Dz));
Axy=sqrt((Ax*Ax)+(Ay*Ay));
Axz=sqrt((Ax*Ax)+(Az*Az));
```

```
Ayz=sqrt((Az*Az)+(Ay*Ay));
Exy=sqrt((Ex*Ex)+(Ey*Ey));
Exz=sqrt((Ex*Ex)+(Ez*Ez));
Eyz=sqrt((Ez*Ez)+(Ey*Ey));
angulo_hombro_xy=(acos((Axy*Axy+Dxy*Dxy-
Exy*Exy)/(2*Axy*Dxy)))*(180/3.1416);
angulo_hombro_xz=(acos((Axz*Axz+Dxz*Dxz-
Exz*Exz)/(2*Axz*Dxz)))*(180/3.1416);
angulo_hombro_yz=(acos((Ayz*Ayz+Dyz*Dyz-
Eyz*Eyz)/(2*Ayz*Dyz)))*(180/3.1416);
```

Figura 19. Cálculo de los ángulos del hombro

Este ángulo calculado no es el que enviaremos al robot pero sí que guarda una relación con él por lo que, como veremos más adelante, lo utilizaremos para calcular el ángulo que finalmente enviaremos al robot.

Sin embargo existe un problema con estos cálculos ya que para diferentes ángulos del hombro la fórmula daría el mismo resultado. Esto se soluciona programando diferentes casos dependiendo de la posición del brazo y si estamos trabajando con el plano XY, XZ o YZ.

Comenzando por el plano XY, como se muestra en la figura 20, dependiendo de si el codo se encuentra por encima o por debajo de la línea que une la base del cuello y el hombro la formula a emplear variará.

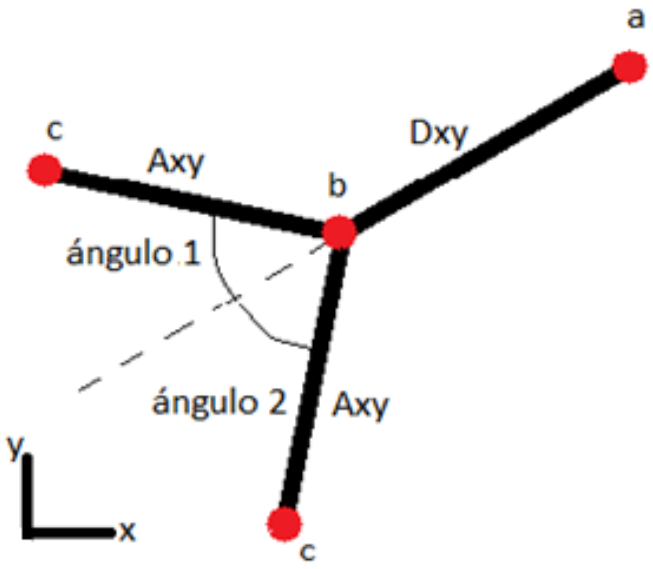

Figura 20. Esquema del a posición del hombro

Lo primero que se hace es calcular la ecuación de dicha recta de la siguiente forma:

$$m_{xy} = \frac{a_y - b_y}{a_x - b_x}$$

$$n_{xy} = a_y - a_x * m_{xy}$$

Donde:

- a_x es la coordenada del punto a en el eje X
- a_y es la coordenada del punto a en el eje Y
- b_x es la coordenada del punto b en el eje X
- b_y es la coordenada del punto b en el eje Y
- m_{xy} es la pendiente de la recta ab en el plano XY
- n_{xy} es la ordenada en el origen de la recta ab en el plano XY

Ahora calculamos la coordenada y del codo si estuviese contenido en esta recta:

$$c_y = c_x * m_{xy} + n_{xy}$$

Donde:

- c_y es la coordenada del punto c en el eje Y si estuviese contenido en la recta ab
- c_x es la cordenada del punto c en el eje X

Si el codo se encuentra por encima de dicha recta la fórmula utilizada sería:

$$angulo\ hombro_{xy_{final}} = 270 - angulo\ hombro_{xy} - angulo\ centro\ hombro\_xy$$

Donde:

- *angulo hombro*$_{xy\ final}$ es el ángulo del hombro del método RPY para el plano XY
- *angulo centro hombro* $_{xy}$ es el ángulo que forma la recta ab con el eje X en el plano XY

Por el contrario, si se encuentra por debajo la fórmula a utilizar será:

$$angulo\ hombro_{xy_{final}} = angulo\ hombro_{xy} - 90 - angulo\ centro\ hombro\_xy$$

Estas fórmulas consideran 0º cuando el brazo se encuentra pegado al cuerpo y 180º cuando el brazo está completamente vertical.

Como se muestra en la fórmula, hay que tener en cuenta también el ángulo formado por la recta que une la base del cuello y el hombro con la horizontal, ya que este no es fijo sino que va disminuyendo a medida que levantamos el brazo y si no lo tenemos en cuenta tendríamos un error en el ángulo del hombro bastante apreciable. Dicha variación de *angulo centro hombro* la podemos apreciar en las figuras 21 y 22 y se calcula de la siguiente forma:

$$angulo\ centro\ hombro\_xy = \text{atan}\left(\frac{a_y - b_y}{b_x - a_x}\right)$$

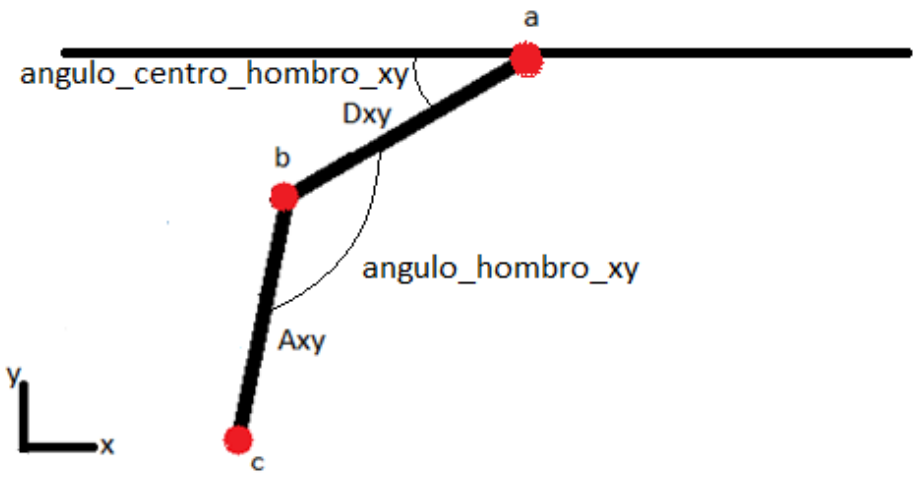

Figura 21. Vista del posicionamiento del brazo en el plano XY

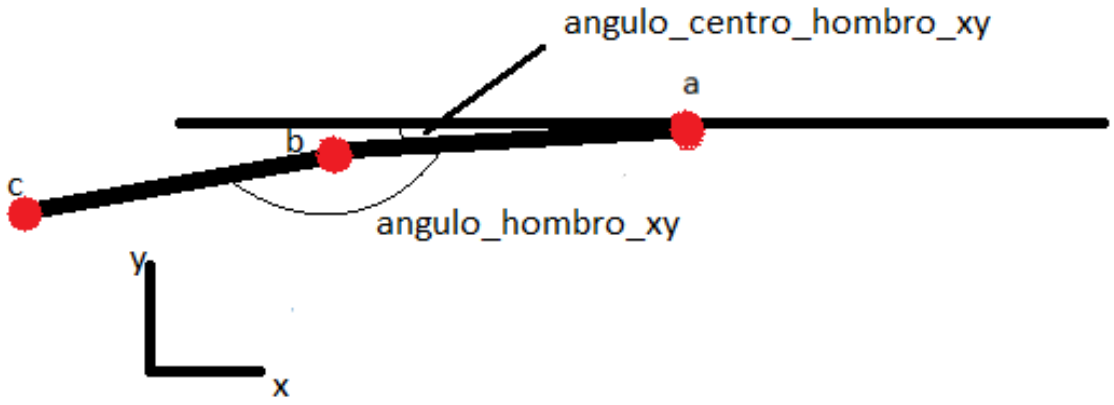

Figura 22. Otro posicionamiento distinto del brazo en el plano XY

A continuación se muestra el código utilizado para realizar las operaciones anteriormente mencionadas:

```
m_xy=(Centroy-hombroy)/(Centrox-hombrox);
n_xy=Centroy-Centrox*m_xy;
angulo_centro_hombro_xy=(atan((Centroy-hombroy)/(hombrox-
Centrox)))*(180/3.1416);
y=codox*m_xy+n_xy;
if(y<=codoy)
{
angulo_hombro_xy=270-angulo_hombro_xy-angulo_centro_hombro_xy;
}
else
      if(y>codoy)
      {
      angulo_hombro_xy=angulo_hombro_xy-90-
angulo_centro_hombro_xy;
      }
```

Figura 23. Cálculo del ángulo final del hombro en el plano XY

Pasamos ahora al plano XZ, en el cual ocurre algo igual a lo explicado anteriormente. Dependiendo de si el codo se encuentra más adelantado o más atrasado que la línea que une la base del cuello con el hombro, usaremos una fórmula u otra. El proceso es igual que el llevado a cabo en el plano XY.

En primer lugar calculamos la ecuación de la recta que une la base del cuello con el hombro de la siguiente forma:

$$m_{xz} = \frac{a_z - b_z}{a_x - b_x}$$

$$n_{xz} = a_z - a_x * m_{xz}$$

Donde:

- a_z es la coordenada del punto a en el eje Z
- b_z es la coordenada del punto b en el eje Z
- m_{xz} es la pendiente de la recta ab en el plano XZ
- n_{xz} es la ordenada en el origen de la recta ab en el plano XZ

Ahora calculamos la coordenada z del codo si estuviese contenido en esta recta así como *angulo centro hombro_xz*, ya que cuando movemos el brazo hacia adelante y hacia atrás la posición relativa del hombro respecto a la base del cuello varía al igual que ocurría en el plano XY. Lo cálculos se realizan como se muestra a continuación:

$$c_z = c_x * m_{xz} + n_{xz}$$

$$angulo\ centro\ hombro\_xz = \operatorname{atan}\left(\frac{a_z - b_z}{b_x - a_x}\right)$$

Donde:

- c_z es la coordenada del punto c en el eje Z si estuviese contenido en la recta ab
- *angulo centro hombro* $_{xy}$ es el ángulo que forma la recta ab con el eje X en el plano XZ

Si el codo se encuentra por detrás de dicha recta la fórmula utilizada sería:

$$angulo\ hombro_{xz_{final}} = 270 - angulo\ hombro_{xz} - angulo\ centro\ hombro\_xz$$

Donde:

- *angulo hombro*$_{xz\ final}$ es el ángulo del hombro del método RPY para el plano XZ
- *angulo hombro*$_{xz}$ es el ángulo que forma la recta ab con la recta bc en el plano XZ

Por el contrario, si se encuentra por delante la fórmula a utilizar será:

$$angulo\ hombro_{xz_{final}} = angulo\ hombro_{xz} - 90 - angulo\ centro\ hombro\_xz$$

Estas fórmulas consideran 0º cuando el brazo se encuentra levantado por delante del cuerpo y paralelo a una recta perpendicular a nuestro tronco y 180º si el brazo se encuentra levantado por detrás del cuerpo y paralelo a una recta perpendicular a nuestro tronco.

A continuación se muestra el código utilizado para realizar las operaciones anteriormente mencionadas:

```
m_xz=(Centroz-hombroz)/(Centrox-hombrox);
n_xz=Centroz-Centrox*m_xz;
angulo_centro_hombro_xz=(atan((Centroz-hombroz)/(hombrox-
Centrox)))*(180/3.1416);
z=codox*m_xz+n_xz;
if(z<=codoz){
angulo_hombro_xz=270-angulo_hombro_xz-angulo_centro_hombro_xz;
}
else
     if(z>codoz){
     angulo_hombro_xz=angulo_hombro_xz-90-
angulo_centro_hombro_xz;
          if(angulo_hombro_xz<0){
          angulo_hombro_xz=0;
          }
     }
```

Figura 24. Cálculo del ángulo final del hombro en el plano XZ

Pasamos ahora al plano YZ, el más complicado de los tres ya que en este plano el hombro puede rotar 360º, por lo que habrá que considerar el problema que teníamos en los planos anteriores pero para más casos.

Como en los casos anteriores comenzaremos calculando la ecuación de la recta que une la base del cuello con el hombro como sigue:

$$m_{yz} = \frac{a_z - b_z}{a_y - b_y}$$

$$n_{yz} = a_y - a_z * m_{yz}$$

Donde:

- m_{yz} es la pendiente de la recta ab en el plano YZ
- n_{yz} es la ordenada en el origen de la recta ab en el plano YZ

Ahora calculamos la coordenada z del codo si estuviese contenido en esta recta así como *angulo centro hombro_yz* como sigue:

$$c_y = c_z * m_{yz} + n_{yz}$$

$$angulo\ centro\ hombro\_yz = \operatorname{atan}\left(\frac{b_z - a_z}{a_y - b_y}\right)$$

Donde:

- c_y es la coordenada del punto c en el eje Y si estuviese contenido en la recta ab
- *angulo centro hombro*$_{yz}$ es el ángulo que forma la recta ab con el eje Y en el plano YZ

Ahora hay que valorar los diferentes casos que nos podemos encontrar y como se calcularía el ángulo del hombro en cada uno de ellos .El primero de ellos es que el codo se encuentre por detrás del hombro, dentro del cual podemos tener tres posibilidades:

- Si el codo se encuentra por debajo de la recta que une la base del cuello con el hombro y la base del cuello se encuentra por delante del hombro la ecuación queda de la siguiente forma:

$$angulo\ hombro_{yz_{final}} = 360 + (180 - angulo\ hombro_{yz} - angulo\ centro\ hombro\_yz$$

Donde:

- *angulo hombro*$_{yz\ final}$ es el ángulo del hombro método RPY para el plano YZ
- *angulo hombro*$_{xy}$ es el ángulo que forma la recta ab con la recta bc en el plano XY

- Si el codo se encuentra por encima de la recta que une la base del cuello con el hombro y la pendiente de dicha recta es mayor que cero la ecuación es la siguiente:

$$angulo\ hombro_{yz_{final}} = 360 - (180 + angulo\ hombro_{yz} + angulo\ centro\ hombro\_yz$$

- Si no se cumplen ninguna de las condiciones anteriores la ecuación a utilizar es:

$$angulo\ hombro_{yz_{final}} = 360 - (180 - angulo\ hombro_{yz} + angulo\ centro\ hombro\_yz$$

El segundo de los casos es que el codo se encuentre por delante del hombro dentro del cual, de nuevo, encontramos tres posibilidades:

- Si el codo se encuentra por debajo de la recta que une la base del cuello con el hombro y la base del cuello se encuentra por detrás del hombro la ecuación queda de la siguiente forma:

$$angulo\ hombro_{yz_{final}} = -(180 - angulo\ hombro_{yz} + angulo\ centro\ hombro\_yz$$

- Si el codo se encuentra por encima de la recta que une la base del cuello con el hombro y la pendiente de dicha recta es menor que cero la ecuación es la siguiente:

$$angulo\ hombro_{yz_{final}} = 180 - angulo\ hombro_{yz} + angulo\ centro\ hombro\_yz$$

- Si no se cumplen ninguna de las condiciones anteriores la ecuación a utilizar es:

$$angulo\ hombro_{yz_{final}} = 180 - angulo\ hombro_{yz} - angulo\ centro\ hombro\_yz$$

Todas las ecuaciones anteriormente mencionadas consideran 0º o 360º cuando el brazo está completamente bajado y pegado a nuestro tronco y sentido antihorario creciente.

A continuación se muestra el código utilizado para realizar las operaciones anteriormente mencionadas:

```
m_yz=(Centroy-hombroy)/(Centroz-hombroz);
n_yz=Centroy-Centroz*m_yz;
angulo_centro_hombro_yz=(atan((hombroz-Centroz)/(Centroy-
hombroy)))*(180/3.1416);
y2=codoz*m_yz+n_yz;
if(codoz>hombroz){
     if((y2>codoy)&&(Centroz<hombroz)){
```

```
      angulo_hombro_yz=360+(180-angulo_hombro_yz-
angulo_centro_hombro_yz);
      }
      else
            if((y2<codoy)&&(m_yz>0)){
            angulo_hombro_yz=360-
(180+angulo_hombro_yz+angulo_centro_hombro_yz);
            }
      else{
      angulo_hombro_yz=360-(180-
angulo_hombro_yz+angulo_centro_hombro_yz);
      }
}
else
      if(codoz<=hombroz){
            if((y2>codoy)&&(Centroz>hombroz)){
            angulo_hombro_yz=-(180-
angulo_hombro_yz+angulo_centro_hombro_yz);
            }
            else
                  if((y2<codoy)&&(m_yz<0)){
                  angulo_hombro_yz=180-
angulo_centro_hombro_yz+angulo_hombro_yz;
                  }
            else{
            angulo_hombro_yz=180-angulo_hombro_yz-
angulo_centro_hombro_yz;
            }
      }
```

Figura 25. Cálculo del ángulo final del hombro en el plano YZ

Una vez calculado el ángulo del hombro en los tres planos pasamos a calcular el ángulo del codo, también en los tres planos. En el caso del codo no nos encontramos con los problemas del hombro ya que le codo solo gira en un sentido por lo que únicamente se explicara el proceso de cálculo para uno de los planos y únicamente habría que repetir el proceso para calcular el resto de ángulos.

Para calcular los ángulos del codo contaremos con tres vértices formados por hombro, codo y muñeca, los cuales forman un triángulo del cual podremos sacar sus lados permitiéndonos así obtener los ángulos mediante el método RPY.

Lo primero que se hace es sacar los lados del triángulo abc del siguiente modo:

$$B_{xy} = \sqrt{B_x^2 + B_y^2}$$

$$C_{xy} = \sqrt{C_x^2 + C_y^2}$$

Donde:

- B_x es la distancia del punto c al d en el eje X
- B_y es la distancia del punto c al d en el eje Y
- C_x es la distancia del punto b al d en el eje X
- C_y es la distancia del punto b al d en el eje Y
- B_{xy} es la distancia del punto c al d en el plano XY
- B_{xy} es la distancia del punto c al d en el plano XY

Ahora utilizamos el teorema del coseno para calcular el ángulo del hombro:

$$angulo\ codo_{xy} = 180 - \arccos\left(\frac{A_{xy}^2 + B_{xy}^2 - C_{xy}^2}{2 * A_{xy} * B_{xy}}\right) * \frac{180}{\pi}$$

Donde:

- *angulo codo*$_{yz}$ es el ángulo del codo del método RPY para el plano XY

A continuación se muestra el código que realiza dichas operaciones para los tres ángulos necesarios:

```
Bx=posicionx-codox;
By=posiciony-codoy;
Bz=posicionz-codoz;
Bxy=sqrt((Bx*Bx)+(By*By));
Bxz=sqrt((Bx*Bx)+(Bz*Bz));
Byz=sqrt((Bz*Bz)+(By*By));
Cx=posicionx-hombrox;
Cy=posiciony-hombroy;
Cz=posicionz-hombroz;
```

```
Cxy=sqrt((Cx*Cx)+(Cy*Cy));
Cxz=sqrt((Cx*Cx)+(Cz*Cz));
Cyz=sqrt((Cz*Cz)+(Cy*Cy));
angulo_codo_xy=180-((acos((Axy*Axy+Bxy*Bxy-
Cxy*Cxy)/(2*Axy*Bxy)))*(180/3.1416));
angulo_codo_xz=180-((acos((Axz*Axz+Bxz*Bxz-
Cxz*Cxz)/(2*Axz*Bxz)))*(180/3.1416));
angulo_codo_yz=180-((acos((Ayz*Ayz+Byz*Byz-
Cyz*Cyz)/(2*Ayz*Byz)))*(180/3.1416));
```

Figura 26. Cálculo del ángulo final del codo

Los ángulos calculados hasta ahora son los ángulos pertenecientes al método RPY, pero cuando se trasladaron estos ángulos al simulador se comprobó que no funcionaban como se esperaba y que en ciertas posiciones el robot no realizaba los movimientos deseados. Ante la falta de tiempo para implementar un nuevo método de para calcular los ángulos articulares se decidió realizar algunas modificaciones a dichos ángulos para que el robot realizase unos movimientos más aproximados a los deseados en dichas posiciones de conflicto. Para ello se introdujo el siguiente código:

```
if(angulo_hombro_xy>180)
{
 angulo_hombro_xy=angulo_hombro_xy-360;
}
if(angulo_hombro_xz>180){
 angulo_hombro_xz=angulo_hombro_xz-360;
}
if(angulo_hombro_yz>180){
 angulo_hombro_yz=angulo_hombro_yz-360;
}
if(angulo_hombro_xy>90){
 angulo_hombro_yz=90-(angulo_hombro_yz-90);
}
if(angulo_hombro_yz<90 && codoy>hombroy){
 angulo_hombro_yz=-angulo_hombro_yz;
}
if(codoz<hombroz && codoy<hombroy && angulo_hombro_xy>45 &&
angulo_hombro_yz>45){
 angulo_hombro_xy=angulo_hombro_xy-angulo_hombro_yz;
}
if(angulo_hombro_xz>45){
```

```
 angulo_codo_final=angulo_codo_xy;
}
 else{
  angulo_codo_final=angulo_codo_yz;
 }
```

Figura 27. Código para la corrección de los ángulos

5. Implementación

Una vez hemos obtenido los ángulos pertinentes, debemos enviarlos en tiempo real al robot humanoide. Sin embargo, lo adecuado es probar el programa en un simulador antes de implementarlo en el robot real para evitar posibles averías. Para realizar este proceso utilizaremos el simulador OpenRAVE instalado en uno de los ordenadores del laboratorio. Dado que este simulador está instalado en Linux y nuestro programa se ejecuta en un PC Windows deberemos pasar los ángulos generados por nuestro programa de una computadora a otra. Para ello usaremos la herramienta YARP y un cable de red cruzado.

5.1. Paso de datos mediante YARP

Para utilizar YARP en nuestro programa deberemos de haber incluido las librerías y archivos descritos en el apartado 4.2. Lo primero hace el programa una comprobación de que exista un *yarp server* activo como se ve en la figura 28, de no ser así nos avisará aunque se podrá proseguir con el resto de funciones.

```
LPCWSTR szAppTitle(_T("Note"));
Network yarp;


if (!yarp.checkNetwork()) {
LPCWSTR szContents = _T("Failed to find YARP server.");
MessageBox( NULL, szContents, szAppTitle, MB_OK | MB_ICONHAND);

}
```

Figura 28. Código de comprobación de YARP

Una vez que hemos obtenido los ángulos definitivos que queremos pasar, abriremos un puerto output llamado *sender* y crearemos dos estructuras flexibles de datos tipo *bottle* que nos permitirán pasar datos de tipo *double*. Para facilitar la posterior recogida de información pasaremos cada ángulo en una línea separada.

```
Bottle codoxy,codoxz,codoyz;
Bottle hombroxy,hombroxz,hombroyz;

hombroxy.addDouble(angulo_hombro_xy);
output.write(hombroxy);
fprintf("angulo hombro XY %s\n", hombroxy.toString().c_str());

hombroxz.addDouble(angulo_hombro_xz);
output.write(hombroxz);
printf("angulo hombro XZ %s\n", hombroxz.toString().c_str());

hombroyz.addDouble(angulo_hombro_yz);
output.write(hombroyz);
fprintf("angulo hombro YZ %s\n", hombroyz.toString().c_str());

codoxy.addDouble(angulo_codo_xy);
output.write(codoxy);
printf("%s\n", codoxy.toString().c_str());

codoxz.addDouble(angulo_codo_xz);
output.write(codoxz);
printf("%s\n", codoxz.toString().c_str());

codoyz.addDouble(angulo_codo_yz);
output.write(codoyz);
printf("%s\n", codoyz.toString().c_str());

}
```

Figura 29. Código del paso de datos

En el ordenador que contiene el simulador, el programa compilado con CMake se encarga de crear un puerto de entrada */reciver* y conectarlo con el puerto */sender* creado anteriormente de modo que los datos escritos en el puerto */sender* sean leídos en el puerto */reciver.* Los datos serán recibidos como se muestra en la figura 30.

```
juanmi@teo-desktop:~$ yarp read /reciver /sender
yarp: Port /reciver active at tcp://163.117.150.58:10022
yarp: Receiving input from /sender to /reciver using tcp
"angulo del eje XY: " 24.544767
"angulo del eje XZ: " 105.677368
"angulo del eje YZ: " -7.304657
"angulo del codo: " 24.949072
"angulo del eje XY: " 24.515934
"angulo del eje XZ: " 107.530273
"angulo del eje YZ: " -8.197845
"angulo del codo: " 24.625254
"angulo del eje XY: " 24.296417
"angulo del eje XZ: " 110.757294
"angulo del eje YZ: " -9.709259
"angulo del codo: " 23.535143
"angulo del eje XY: " 23.624346
"angulo del eje XZ: " 116.071999
"angulo del eje YZ: " -12.080292
"angulo del codo: " 21.321417
"angulo del eje XY: " 22.824495
"angulo del eje XZ: " 119.111298
"angulo del eje YZ: " -13.199341
"angulo del codo: " 18.693098
"angulo del eje XY: " 22.165184
```

Figura 30. Prueba de recepción

5.2. Conexión de los ordenadores

Dado que en un trabajo anterior en el que había que realizar una transmisión de datos parecida a la que había que realizar en este caso se usó un cable de red cruzado, se decidió usarlo también en este ya que se sabía que era un buen método y funcionaba correctamente. Este cable va conectado a las entradas usuales para conexión de datos e Internet de ambos ordenadores.

Ahora hay que configurar las IP's y máscaras de subred de ambos ordenadores para que sean capaces de comunicarse. Para ello accederemos a las opciones de configuración de red de ambos ordenadores y cambiaremos la IP y la máscara de subred. La máscara de subred debe ser la misma en ambos ordenadores y la IP debe coincidir en los tres primeros números, variando únicamente el cuarto.

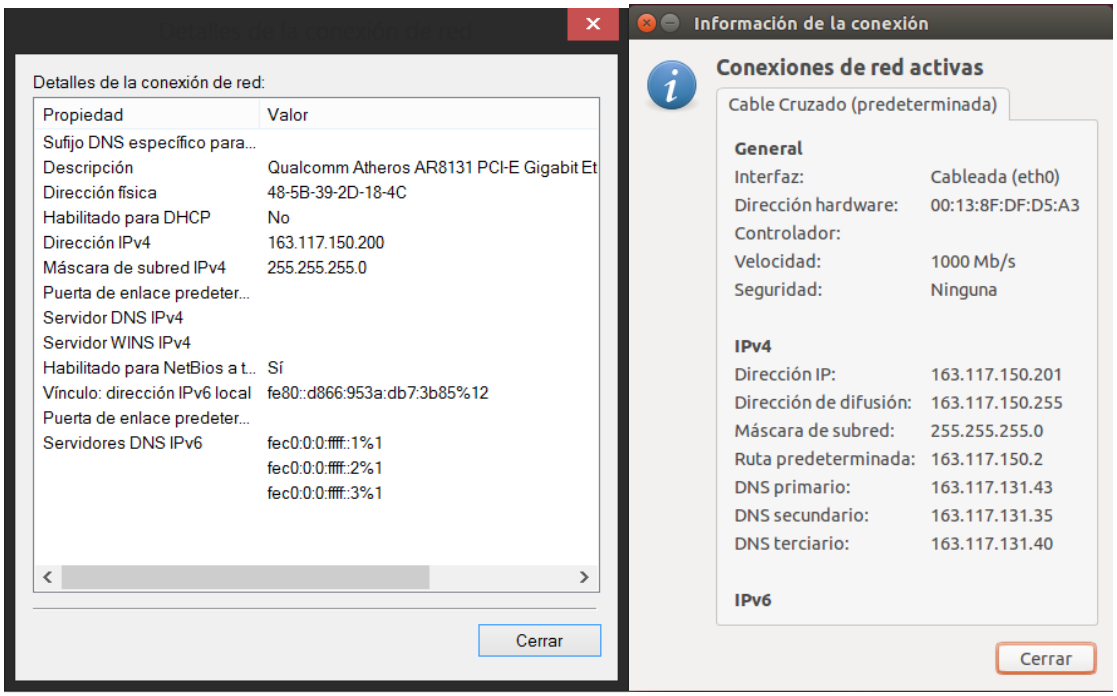

Figura 31. Conexiones de área local

En nuestro caso será 163.117.150.200 para el ordenador que manda la información y 163.117.150.201 para el que la recibe como se muestra en la figura 31. Será necesario que en uno de los ordenadores haya un *yarp server* activo en el que se comprobará si la conexión se ha efectuado correctamente. Todos los datos escritos en el puerto *sender* se verán automáticamente reflejados en el ordenador asociado al puerto *reciver.* Cuando se ejecuta el *yarp server* por primera vez por defecto se crea un puerto */root.* Dado que en este caso el *yarp server* se ejecutó en el ordenador que enviaba la información, dicho puerto fue creado en la IP acabada en .200. Podemos ver en las figuras 32 y 33 como posteriormente y una vez ejecutados los programas en ambos ordenadores se crean sendos puertos *sender* y *reciver* y se conectan entre sí.

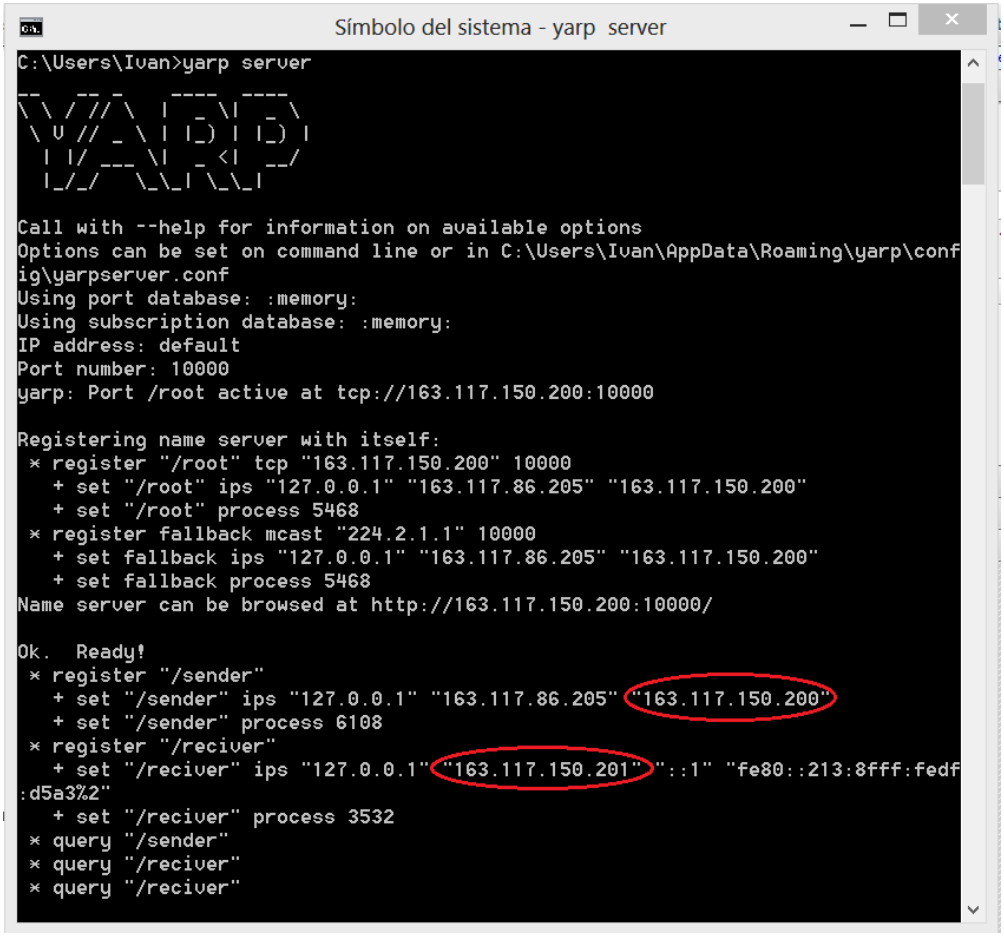

Figura 32. Creación de los puertos sender y reciver

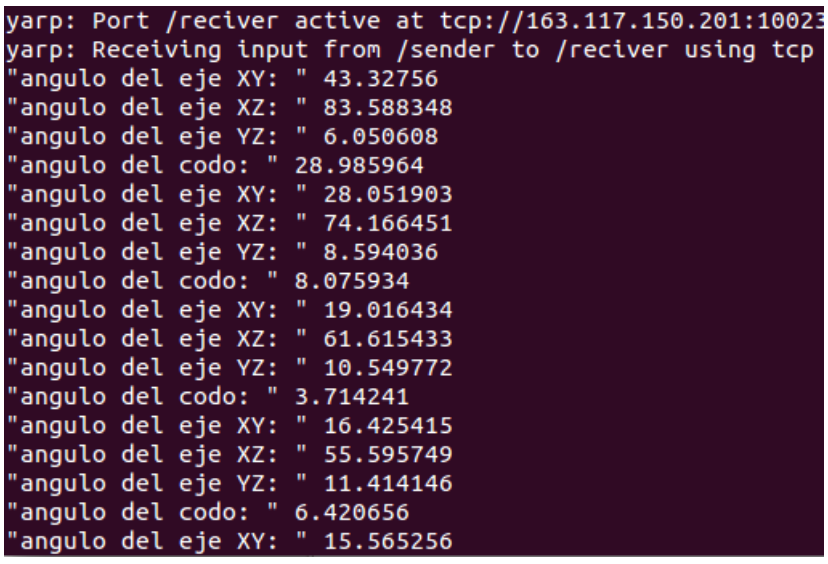

Figura 33. Conexión de los puertos

Universidad Carlos III de Madrid

5.3. Simulación

Aunque este trabajo está pensado para su implementación en un robot humanoide, antes de llevar a cabo este paso y por motivos de seguridad se decidió probarlo en un simulador, en nuestro caso OpenRAVE. Este tipo de pruebas son muy comunes y prácticamente imprescindibles en cualquier proyecto que implique un programa en desarrollo que todavía no ha sido probado con el fin de corregir posibles errores que pudieran surgir sin poner en peligro la integridad del robot humanoide. Para este proyecto se usará el módulo *teoSim* que contiene una versión completa y realista del robot TEO, aunque nosotros nos centraremos en la simulación del brazo derecho.

Lo primero que haremos será instalar el simulador y las bibliotecas necesarias. Puesto que el sistema operativo utilizado en este caso es Linux pero la programación se sigue realizando en C++, se pensó que sería un buen aprendizaje aprender a utilizar CMake para realizar la compilación y creación del ejecutable.

Ya sea partiendo de alguno de los ejemplos instalados con OpenRave, deberemos de crear el programa en un archivo con extensión .cpp del modo que se muestra en la memoria. Una vez esté completo crearemos un archivo "CMakeLists.txt" con los datos que aparecen al final de este apartado, el cual se deberá encontrar en el mismo directorio que el archivo .cpp.

Ahora abrimos una terminal de Linux y nos situamos en el directorio en el que se encuentran los dos archivos creados anteriormente y escribiremos "mkdir build" lo que creará una carpeta llamada "build" que contendrá los archivos generados en la compilación del programa mediante CMake. Una vez creada dicha carpeta nos situamos dentro de ella y ejecutamos el comando "CMake", lo que creará varios archivos, entre los que se encontrará un "MakeFile", dentro de la carpeta "build" necesarios para la posterior creación del ejecutable de nuestro programa. A continuación escribimos "make", lo que creará el ejecutable de nuestro programa también dentro de la carpeta "build" con el nombre que hayamos especificado en el archivo "CMakLists.txt", en nuestro caso "testTEO".

Antes de ejecutar el programa necesitamos tener abierto el entorno de simulación para lo que abrimos una terminal nueva, nos situaremos dentro de la carpeta "bin" dentro de la cual se encuentra el ejecutable del paquete de simulación del robot TEO e introducimos el comando "teoSim". Se abrirá una ventana con una simulación del robot y en la "terminal" en la que lo hemos abierto y no debemos cerrar hasta terminar con el simulador aparecerá información referente a este.

Universidad Carlos III de Madrid

Una vez abierto el entorno de simulación ejecutaremos el programa. Lo primero que hará el programa será comprobar que exista un *yarp server* y alguno de los simuladores de RaveBot esté activo, si cualquiera de los dos fallara el programa se detendría. Una vez comprobado establece la conexión con el simulador y crea un puerto *receiver* y lo conectará a *sender* ya que será el encargado de recibir los datos procesados anteriormente. Todo este proceso se realiza mediante el código mostrado a continuación:

```
Network yarp;
if(!Network::checkNetwork())
{

     printf("Please start a yarpname server first\n");
     return(-1);

}
Port input;
Property options;
options.put("device","remote_controlboard");
options.put("remote","/teoSim");
options.put("local","/local");
PolyDriver dd(options);
```

```
if(!dd.isValid())
{

     printf("RaveBot device not available.\n");
     dd.close();
     Network::fini();
     return1;
}
```

Figura 34. Comprobación, creación y conexión de puertos

Llegados a este punto, el brazo simulado será llevado a la posición de inicio antes de comenzar a pasar más instrucciones. Una vez esto se empezará a valorar los datos que llegan, los cuales vienen en formato *string*, por lo que utilizaremos la sentencia atof para transformarlos en un *double* y poder operar con ellos.

```
// se coloca en posición de inicio
pos->positionMove(0, -90);
pos->positionMove(1, 0);
pos->positionMove(2, 0);
pos->positionMove(3, 0);

Bottle bot;
input.read(bot);
dato=atof(bot.toString().c_str());//paso de string a double
```

Figura 35. Comienzo de la lectura de los datos recibidos

Los datos vendrán en orden comenzando por el ángulo del hombro en el plano XY, seguido del ángulo del hombro en el plano XZ, el ángulo del hombro en el plano YZ y, por último, el ángulo del codo, tomados todos en el mismo instante. Estos datos serán pasados al simulador estipulando el número de eje que se quiere mover (1 para el hombro en el plano XY, 2 para el hombro en el plano XZ, 0 para el hombro en el plano YZ y 3 para el codo) y el valor del ángulo que se quiere mover. Por ejemplo:

Pos->positionMove (eje 1, angulo_hombro);

Para realizar las pruebas y comprobar que el movimiento del simulador se correspondía con los ángulos enviados se mostraba por pantalla los datos enviados al simulador.

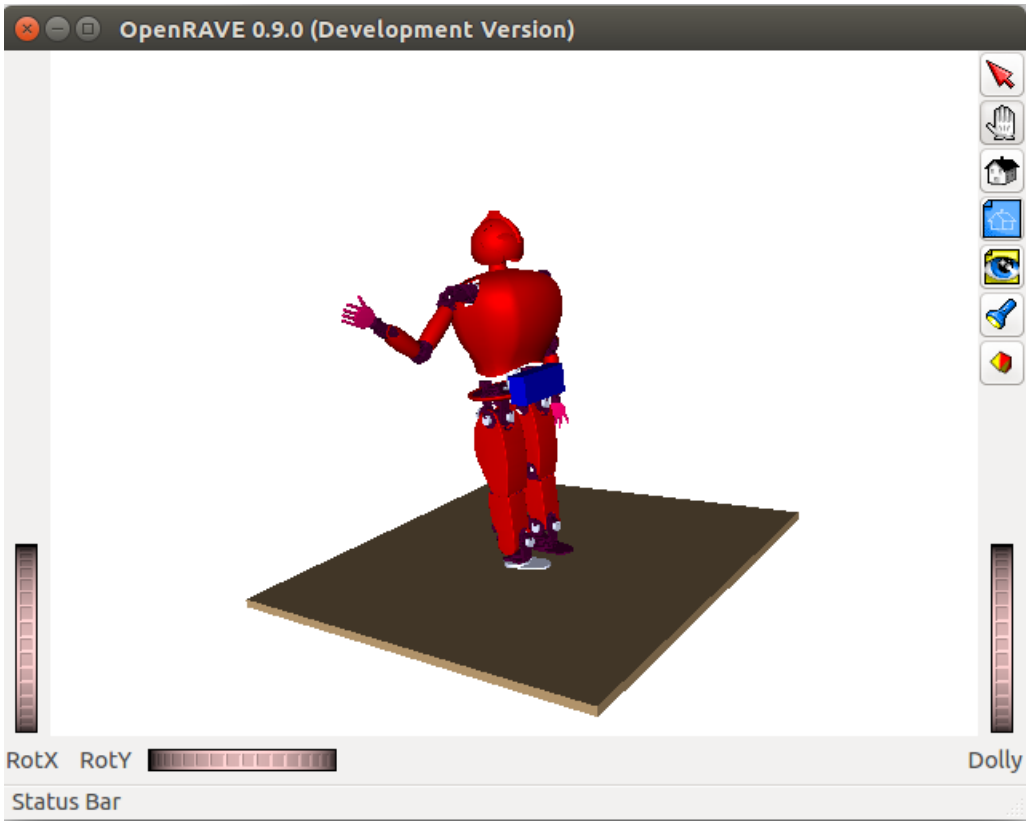

Figura 36. Robot simulado

```
test positionMove(1, 50.821198)
El angulo del hombro XZ es 93.673111
test positionMove(2, 93.673111)
El angulo del hombro YZ es -4.504089
test positionMove(0, -4.504089)
El angulo del codo es 76.347008
test positionMove(3, 76.347008)
El angulo del hombro XY es 50.385853
test positionMove(1, 50.385853)
El angulo del hombro XZ es 93.914711
test positionMove(2, 93.914711)
El angulo del hombro YZ es -4.726532
test positionMove(0, -4.726532)
El angulo del codo es 76.872185
test positionMove(3, 76.872185)
El angulo del hombro XY es 50.070908
test positionMove(1, 50.070908)
El angulo del hombro XZ es 94.093948
test positionMove(2, 94.093948)
El angulo del hombro YZ es -4.887878
test positionMove(0, -4.887878)
El angulo del codo es 77.253021
test positionMove(3, 77.253021)
```

Figura 37. Datos recibidos y enviados al simulador

Como se puede comprobar en la figura 36, la posición en la que se encuentra el simulador se corresponde con los datos enviados, esto es, el hombro abierto unos 50º en el plano XY, unos 90º en el plano XZ, casi 0º en el plano YZ y con el codo doblado casi 80º. Como se muestra en la figura 37, los datos enviados al simulador se corresponden con la posición real de la persona.

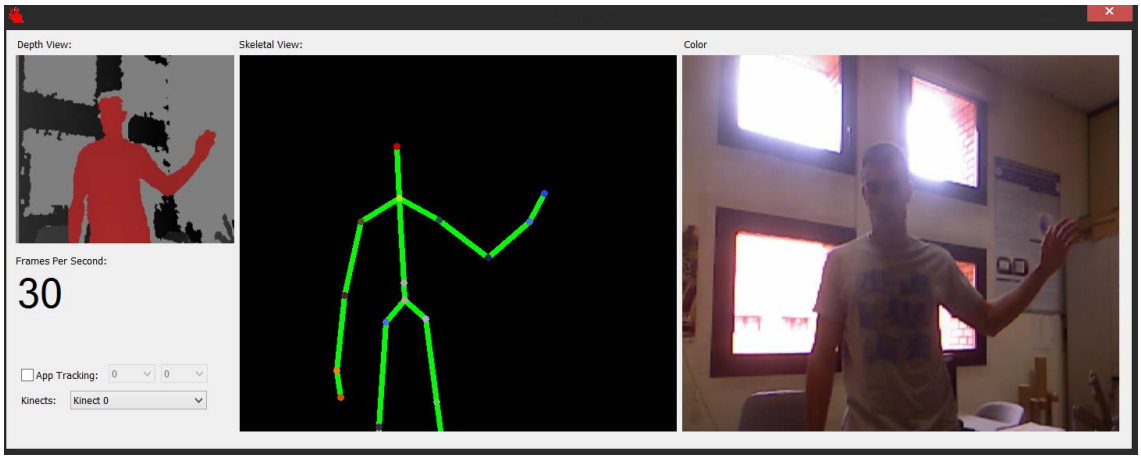

Figura 38. Imagen real

Independientemente de las órdenes que reciba por parte de nuestro programa, al simulador lo dotaremos de unas condiciones propias que no será posible sobrepasar. De este modo estableceremos una velocidad máxima a la que puede rotar cada eje sin que pusiera en peligro al robot real y unos ángulos máximo que puede alcanzar en la rotación basándose en los límites físicos del robot humanoide.

Por último creamos un archivo llamado CMakeList.txt en el mismo directorio que el documento .cpp que contiene el código descrito anteriormente. El archivo de texto contendrá las siguientes líneas:

- cmake_minimum_required(VERSION 2.6) → Si la versión es anterior a la 2.6 se negará la compilación.
- set(KEYWORD "testTEO"); project(${KEYWORD}) → Crea un proyecto con el nombre “testTEO”.
- find_package(YARP REQUIRED) → Comprueba que esté instalado Yarp.
- find_package(TEO REQUIRED) → Comprueba que exista el paquete teoSim para el uso del simulador.
- include_directories(${YARP_INCLUDE_DIRS} ${TEO_INCLUDE_DIRS})
- Enlaza los archivos de inclusión de Yarp.
- link_directories(${TEO_LINK_DIRS})
- add_executable(testTEO main.cpp) → Señala el archivo .cpp a partir del cual deberemos de compilar y crear el ejecutable testTEO.
- target_link_libraries(testTEO ${YARP_LIBRARIES} ${TEO_LIBRARIES}) → Incluye las librerias de Yarp y de teoSim en la compilación del archivo.

6. Análisis de resultados

Habiendo explicado los pasos que hemos seguido para completar el proyecto llega el momento de evaluar los resultados obtenidos para ver la fiabilidad del programa. Comenzaremos por guardar los datos de los ángulos a un documento de texto para poder pasarlos a una tabla Excel y hacer gráficas en las que podamos valorar el movimiento descrito.

En el primer experimento evaluaremos el ángulo del hombro en los tres planos XY, XZ e YZ en ese orden. Para evaluar el ángulo del hombro en el plano XY describiremos un movimiento constante de subida y bajada cuya trayectoria se puede apreciar en las figura 39 a 42. Comenzaremos con el brazo abajo formando unos 10º con la vertical y llegaremos al máximo que nos permita el hombro que como se puede observar en la figura 45 es de aproximadamente 150º. La medición está tomada a lo largo de aproximadamente 26,9 segundos en los que se tomaron un total de 807 muestras a una velocidad de 30 fps.

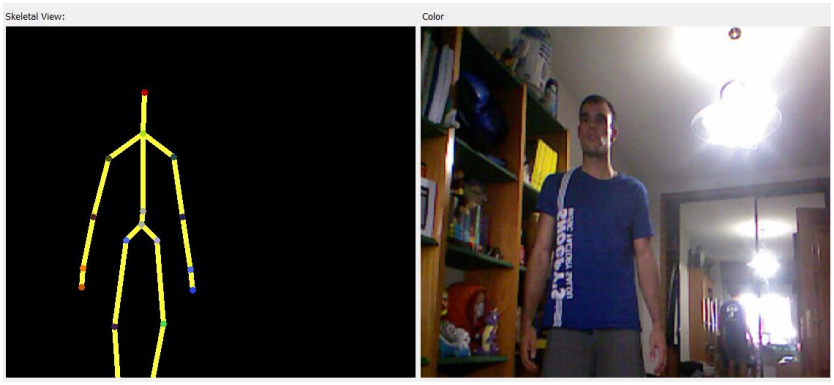

Figura 39. Movimiento del hombro en el plano XY (1)

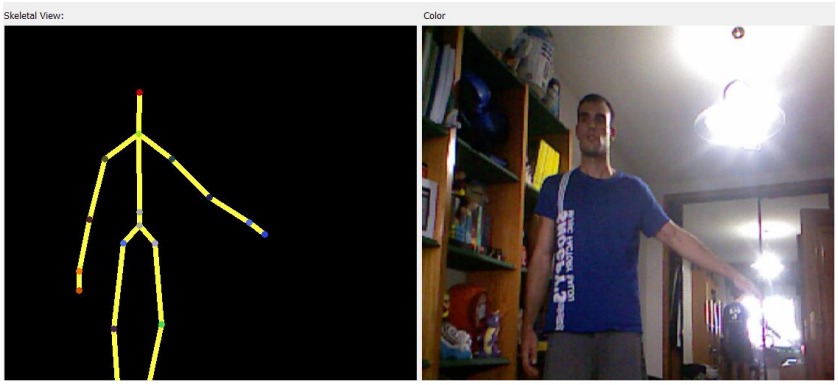

Figura 40. Movimiento del hombro en el plano XY (2)

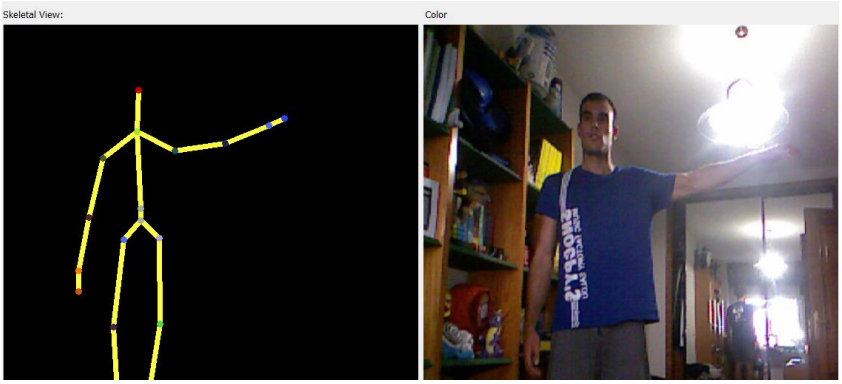

Figura 41. Movimiento del hombro en el plano XY (3)

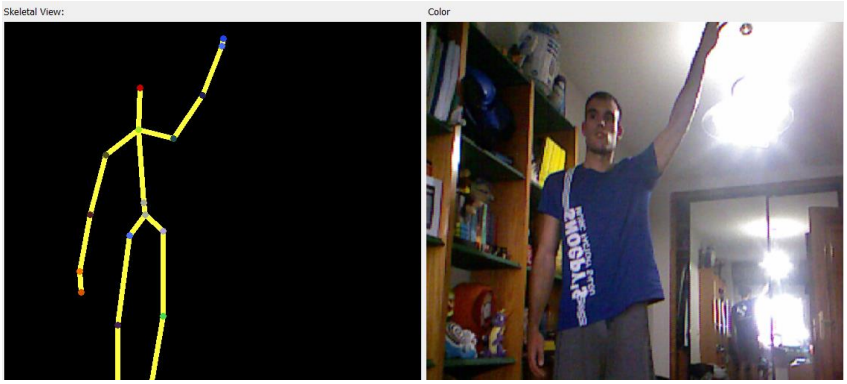

Figura 42. Movimiento del hombro en el plano XY (4)

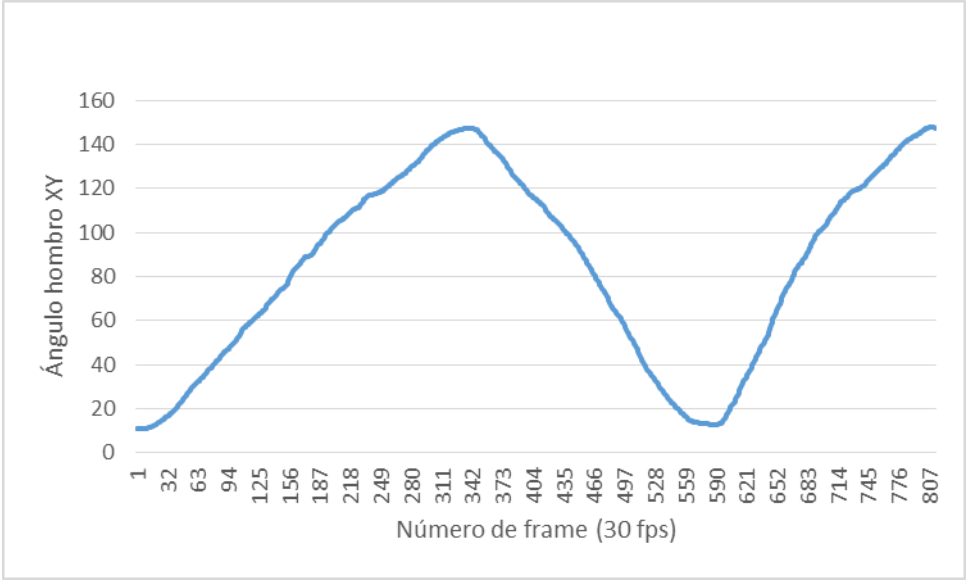

Figura 43. Ángulo hombro XY

Como se puede apreciar en la figura 43, el movimiento recogido se ajusta bastante bien a un movimiento suave de subida y bajada del brazo. Las pequeñas variaciones que se observan se corresponden a la incapacidad física de la persona de llevar una velocidad constante, lo que se comprueba también al observar cómo tanto el movimiento de bajada como el segundo movimiento de subida son más rápidos que el primer movimiento de subida.

Pasaremos ahora a evaluar el ángulo del hombro en el plano XZ, para lo que comenzaremos con el brazo completamente estirado y perpendicular al tronco formando unos 90º con una línea perpendicular al tronco. Posteriormente avanzaremos el brazo hasta que quede prácticamente paralelo a la línea perpendicular al tronco haciendo que el ángulo del hombro llegue casi hasta los 0º para después llevarlo hacia atrás tanto como nos permita la articulación y, por último, volver a moverlo hacia adelante como hicimos en el primer movimiento. Dicha trayectoria se puede observar en las figuras 44 a 46.

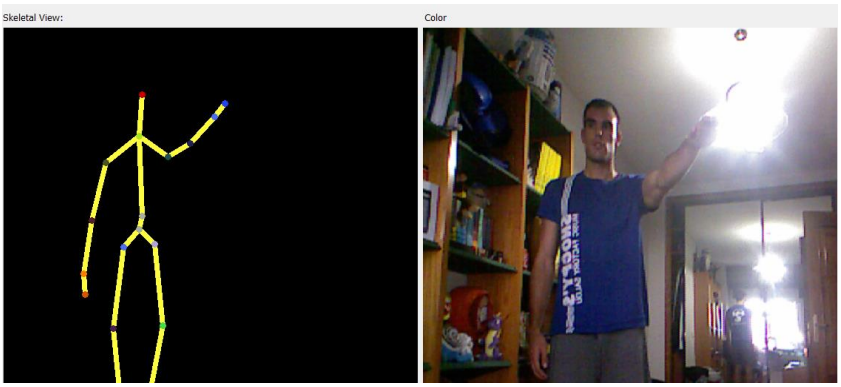

Figura 44. Movimiento del hombro en el plano XZ (1)

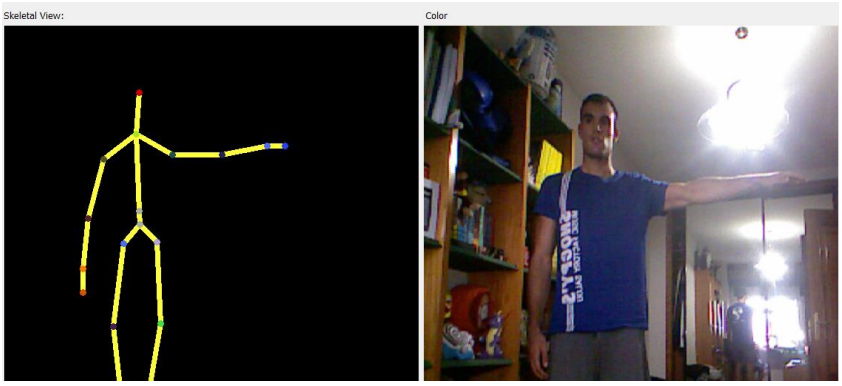

Figura 45. Movimiento del hombro en el plano XZ (2)

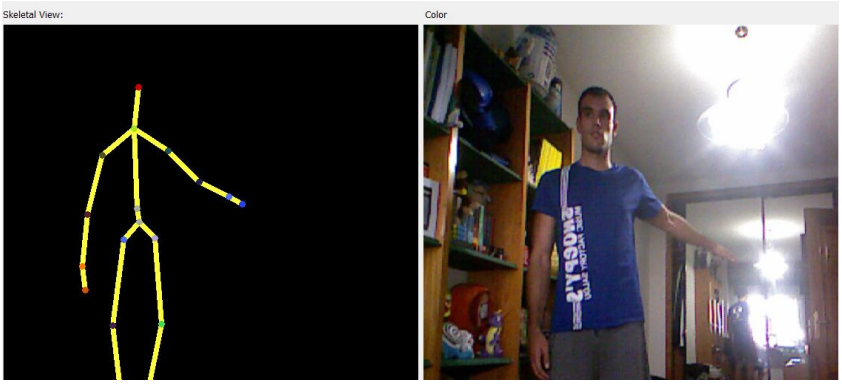

Figura 46. Movimiento del hombro en el plano XZ (3)

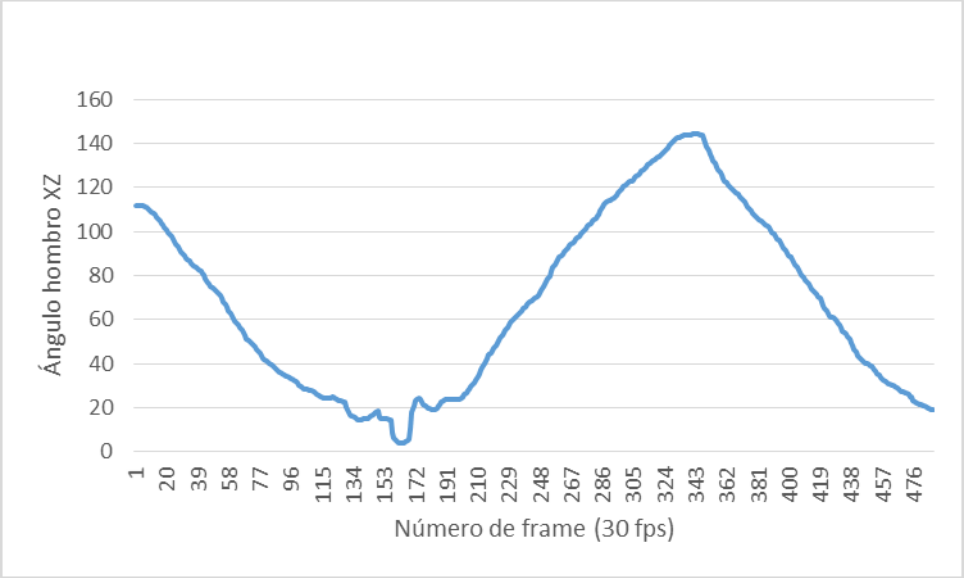

Figura 47. Ángulo hombro XZ

Como se puede observar en la figura 47, este movimiento también se corresponde bastante bien con el movimiento descrito anteriormente y, de nuevo, las pequeñas variaciones se corresponden con la incapacidad de la persona de llevar una velocidad constante. Sin embargo, en este caso se puede apreciar otra irregularidad mucho mayor cuando el brazo se encuentra cerca de estar perpendicular al tronco, lo cual es debido a que en estas posiciones, los puntos que representan el codo, el hombro y la base del cuello se encuentran muy cerca unos de otro y, en ocasiones, llegan a superponerse haciendo muy difícil al programa distinguir una de otra.

Para evaluar el la captura del ángulo del hombro en el plano YZ, la trayectoria seguida se puede observar en las figuras 48 a 51 consistirá en dos

giros completos con el brazo estirado, en sentido antihorario, y comenzando con el brazo pegado al tronco, por lo que se comenzará en un ángulo aproximado de 0º que irá creciendo hasta los 360º y de nuevo volverá a 0º al comenzar la segunda vuelta.

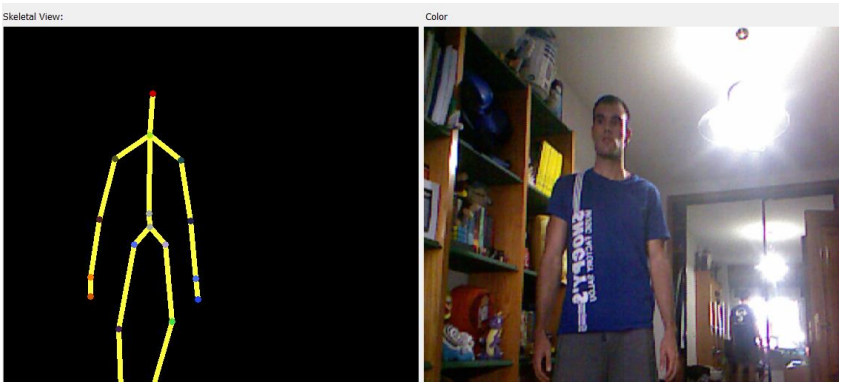

Figura 48. Movimiento del hombro en el plano YZ (1)

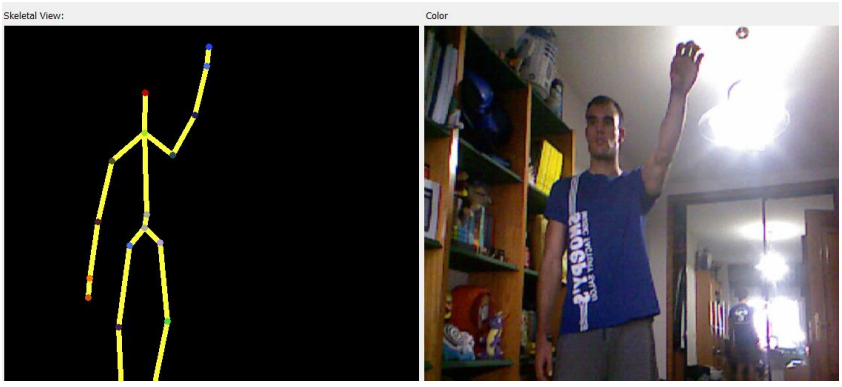

Figura 49. Movimiento del hombro en el plano YZ (2)

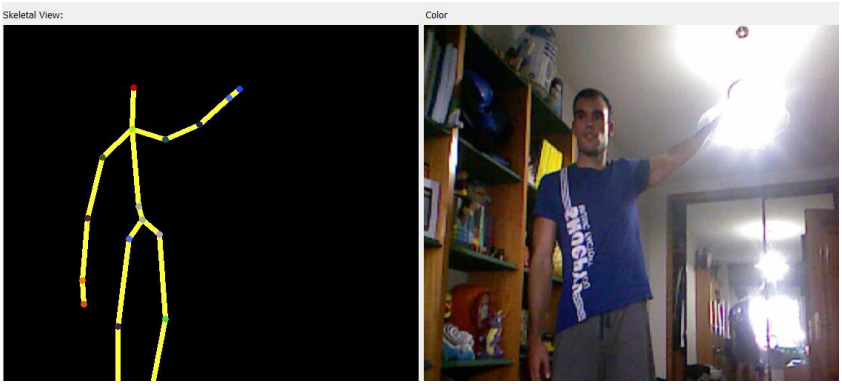

Figura 50. Movimiento del hombro en el plano YZ (3)

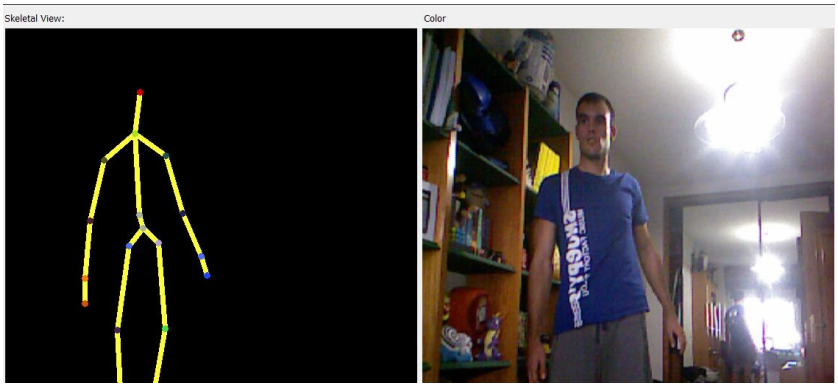

Figura 51. Movimiento del hombro en el plano YZ (4)

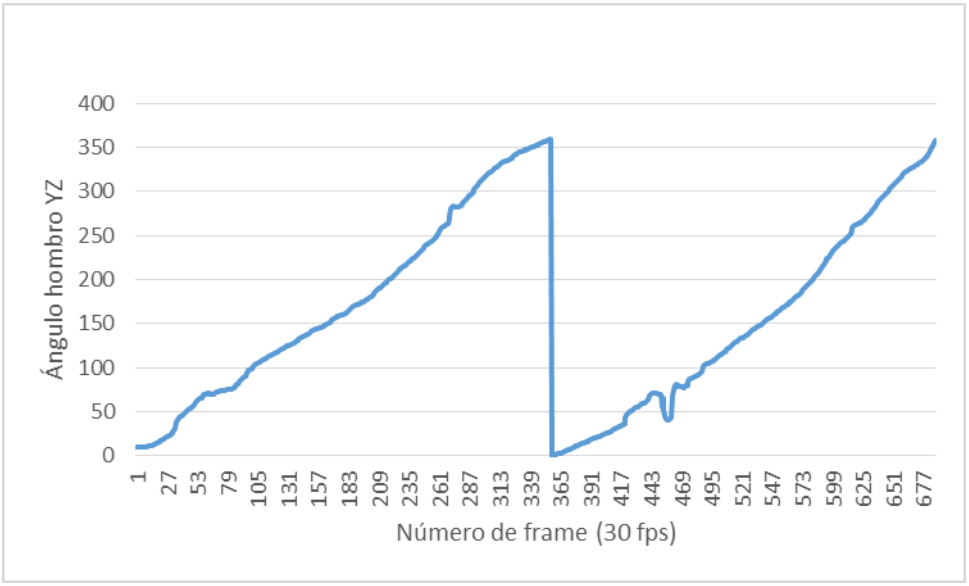

Figura 52. Ángulo hombro YZ

De nuevo, en la figura 52 se aprecia que el movimiento se corresponde bastante bien con el movimiento descrito, pudiéndose apreciar pequeñas irregularidades al no ser capaz de mantener una velocidad constante. También podemos ver algunas irregularidades mayores debidas al fenómeno comentado anteriormente por el que los puntos que representan las articulaciones se encuentran muy juntos unos de otros haciendo muy difícil al programa poder distinguirlos, dando lugar a ángulos erróneos que provocan estas variaciones bruscas. Por último se puede apreciar claramente el momento en el que el brazo pasa de encontrarse más atrasado que el tronco a encontrarse más adelantado y, por lo tanto, pasando el ángulo de 360º a 0º.

En la siguiente prueba valoraremos un movimiento constante de flexión y extensión del antebrazo con el hombro aproximadamente inmóvil tal como aparece en las figuras 53 a 55. Partiremos de un brazo casi estirado que el programa reconoce por un ángulo de unos 25º e iremos flexionando el codo hasta llegar a los 150º, o lo que es lo mismo, un ángulo de 30º entre el bíceps y el antebrazo.

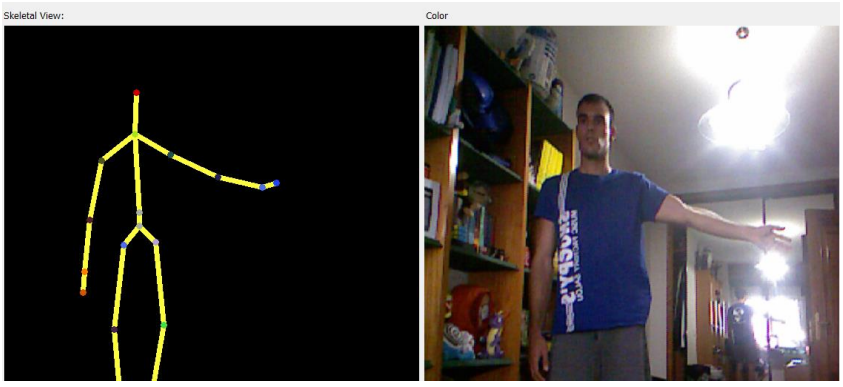

Figura 53. Movimiento del codo (1)

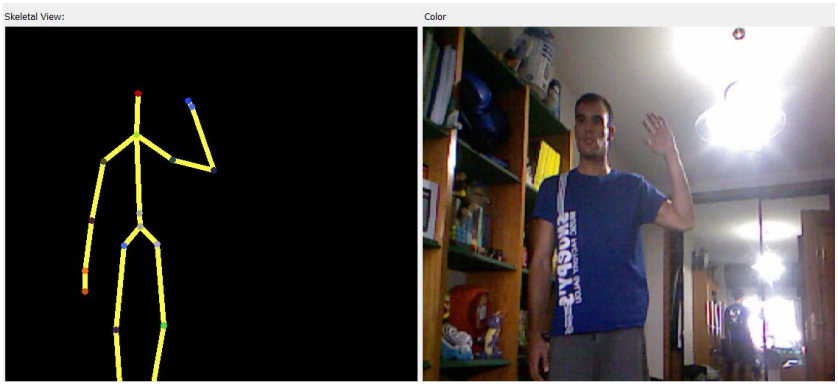

Figura 54. Movimiento del codo (2)

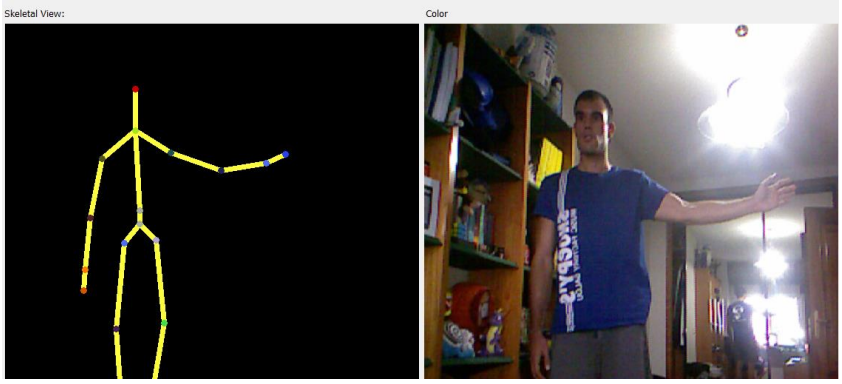

Figura 55. Movimiento del codo (3)

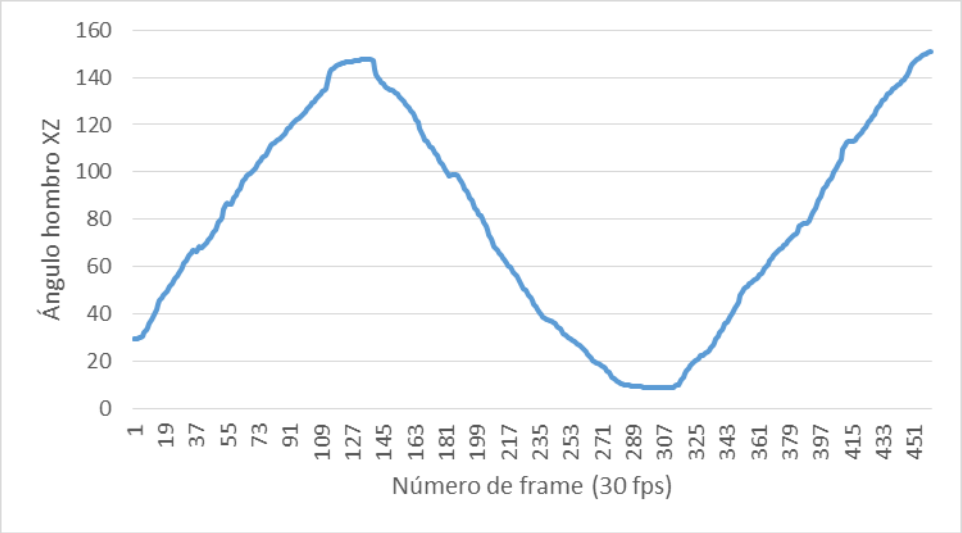

Figura 56. Ángulo codo

La gráfica de la figura 56 se sigue ajustando bastante a un movimiento constante aunque vemos que varía igual que en los experimentos anteriores, sobretodo en el punto de flexión máxima que corresponderá al entorno de los 140º. Es en estos momentos donde el programa tiene más problemas al hacer la esqueletización ya que hay muy poca separación entre las dos partes de la extremidad y el punto de la muñeca será más complicado de situar como se explicó anteriormente. Al mismo tiempo, elementos como la ropa distorsionan la figura del cuerpo y dificultan la labor del programa.

Por último se realizara un experimento en el que se evaluarán los tres ángulos del hombro al mismo tiempo para comprobar que dichos ángulos se corresponden con la realidad. Para ello se empezará con el brazo pegado al tronco y se elevará de forma oblicua hasta que quede paralelo al suelo, es decir, que los ángulos de los planos XY e YZ están a unos 90º, momento en el que se moverá hacia adelante, hacia atrás, y de nuevo hacia adelante hasta quedar en la posición inicial paralelo al suelo, para posteriormente elevarlo pero sin llegar a forzar la articulación. La trayectoria de este movimiento se puede observar en las figuras 57 a 63.

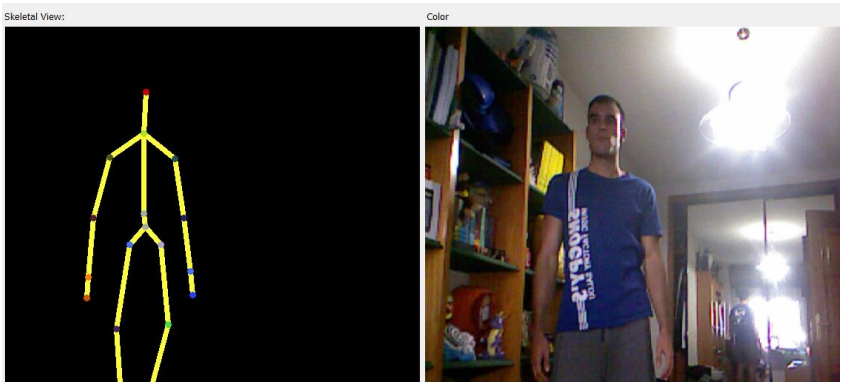

Figura 57. Trayectoria del hombro en movimiento compuesto (1)

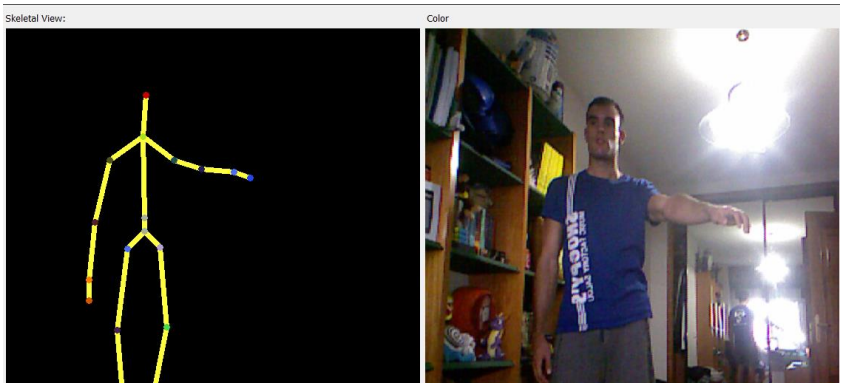

Figura 58. Trayectoria del hombro en movimiento compuesto (2)

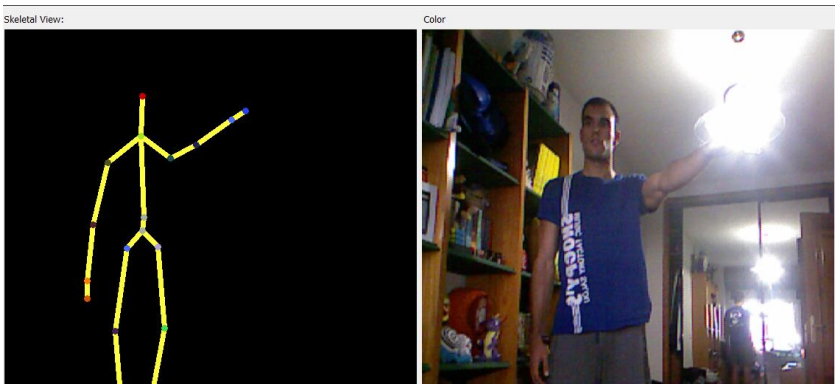

Figura 59. Trayectoria del hombro en movimiento compuesto (3)

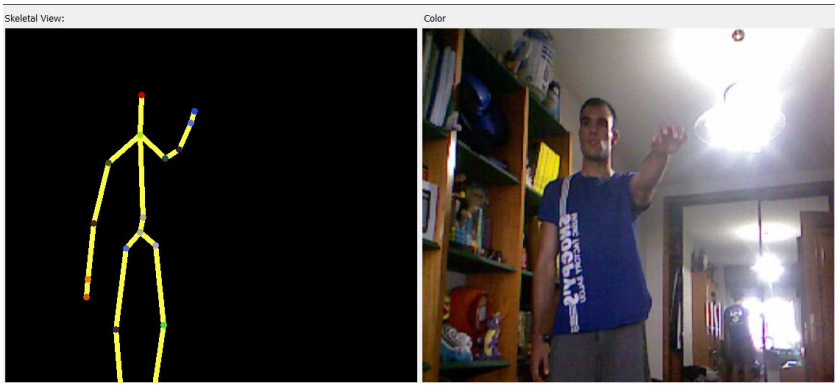

Figura 60. Trayectoria del hombro en movimiento compuesto (4)

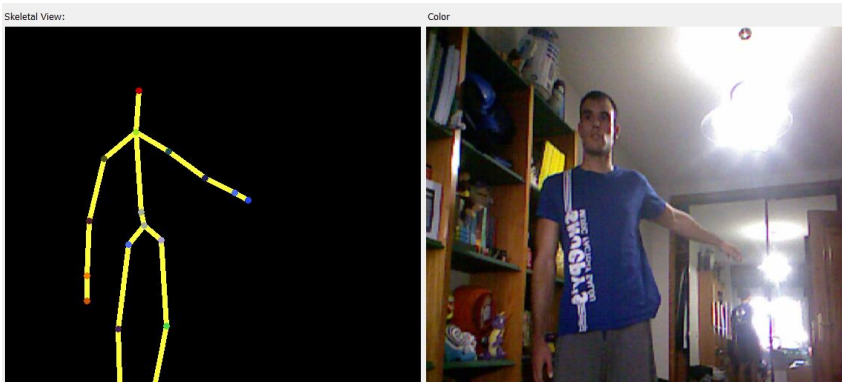

Figura 61. Trayectoria del hombro en movimiento compuesto (5)

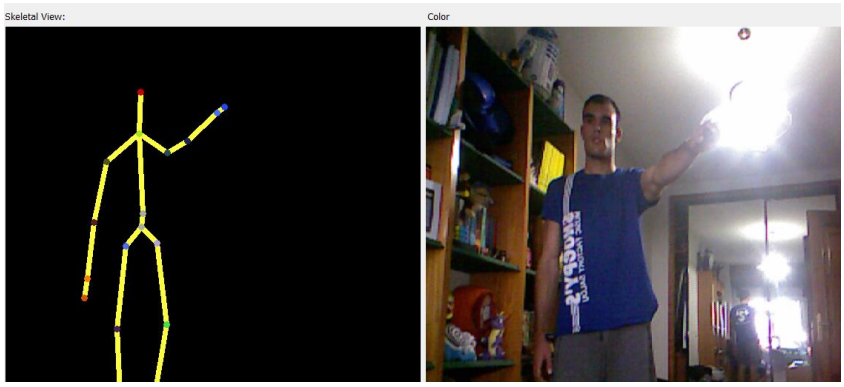

Figura 62. Trayectoria del hombro en movimiento compuesto (6)

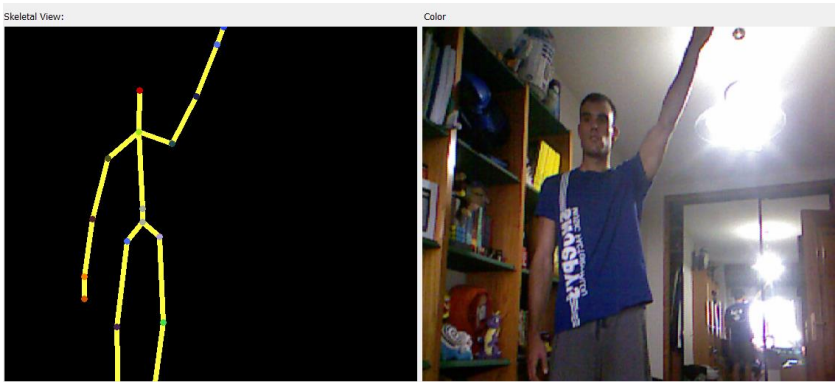

Figura 63. Trayectoria del hombro en movimiento compuesto (7)

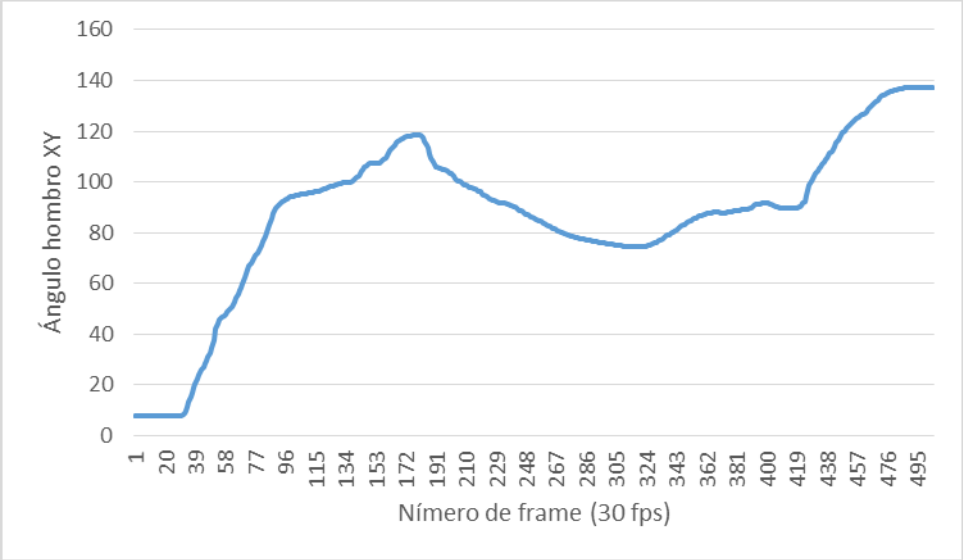

Figura 64. Ángulo hombro XY en movimiento compuesto

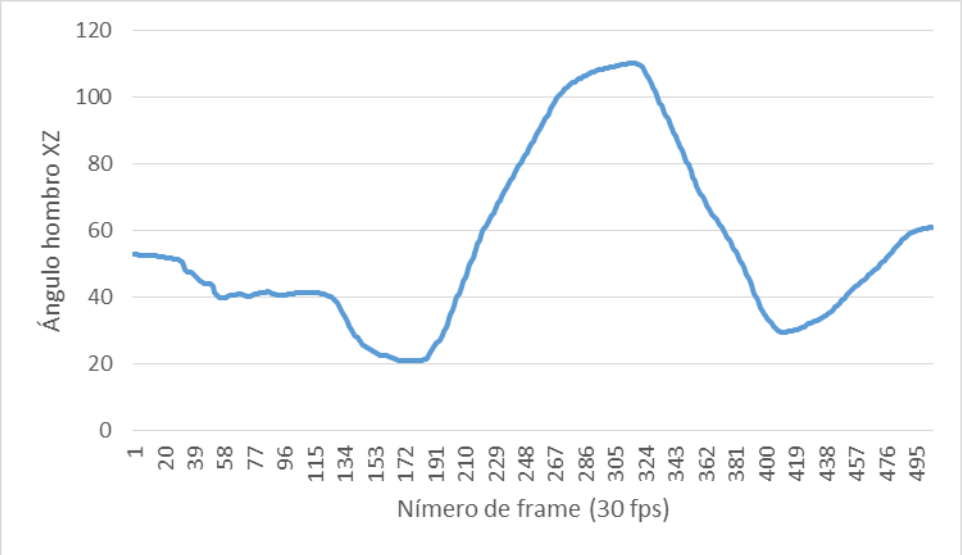

Figura 65. Ángulo hombro XZ en movimiento compuesto

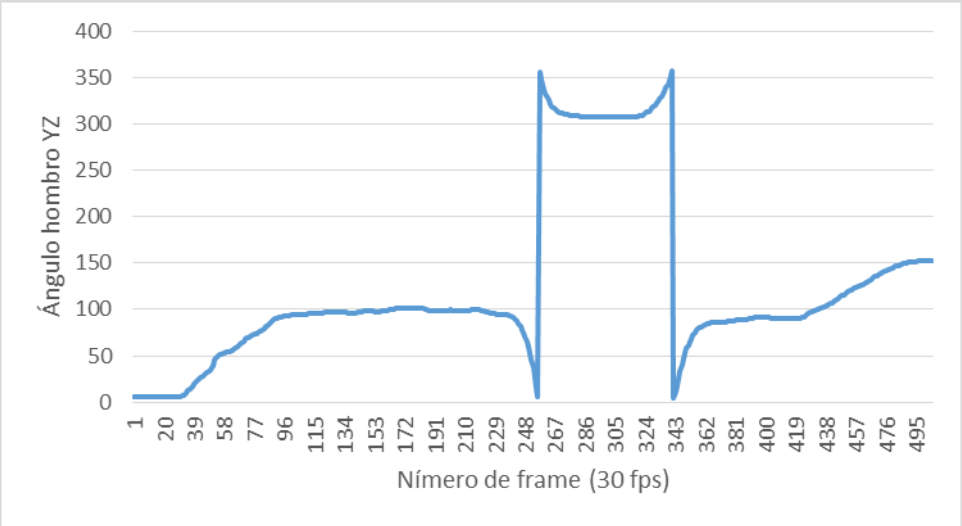

Figura 66. Ángulo hombro YZ en movimiento compuesto

Comparando las tres gráficas se puede apreciar el movimiento descrito anteriormente aunque con bastantes irregularidades ante la dificultad física de, por ejemplo, realizar movimientos paralelos al suelo manteniendo la misma altura.

Fijándonos en la gráfica 64 que describe el movimiento del hombro en el plano XY podemos observar como partimos de casi 0º, es decir, el brazo

pegado al tronco y como a medida que levantamos el brazo el ángulo va aumentando hasta alcanzar la posición deseada rondando los 90º. Es cierto que el ángulo fluctúa bastante, lo cual es debido a que al realizar los movimientos horizontales indicados anteriormente es difícil mantener la altura estable. Finalmente el ángulo vuelve a aumentar cuando levantamos el brazo para terminar el movimiento.

Respecto a la gráfica 65 que representa el movimiento del hombro en el plano XZ, podemos observar que mientras elevamos el brazo de forma oblicua, el ángulo se mantiene próximo a los 45º, que sería el ángulo deseado. Posteriormente realizamos el movimiento hacia adelante con el brazo paralelo al suelo llegando hasta los 20º, luego hacia atrás llegando a los 110º y finalmente tratamos de volver a una posición de 45º aunque, como se aprecia en la gráfica, nos pasamos y llegamos hasta los 25º. Finalmente, al elevar el brazo tratamos de corregir la desviación anterior pasándonos esta vez y llegando hasta los 60º con el brazo ya levantado.

Por último, fijándonos en la figura 66 que representa el movimiento del hombro en el plano YZ, se ve como partimos de los 0º que representan el brazo pegado al torso y éstos van aumentando a medida que levantamos el brazo hasta estabilizarse en los 100º mientras realizamos los movimientos horizontales. Además, si comparamos las figuras 65 y 66 podemos ver cómo en el momento en el que el ángulo del hombro en el plano XZ pasa de los 90º, es decir, el brazo pasa de estar por delante del torso a estar por detrás, el ángulo del hombro en el plano YZ pasa rápidamente de los 100º a casi los 310º. Si el movimiento horizontal fuese perfecto el ángulo debería pasar de 90º a 270º, sin embargo llegamos casi a los 310º debido a que los humanos al mover el brazo hacia atrás paralelo al suelo no podemos y tendemos a bajar el brazo para llegar lo más atrás posible. Cuando el brazo vuelve a estar por delante del torso el ángulo vuelve en este caso a los 90º y a partir de ahí vemos como a medida que levantamos el brazo en el último movimiento, el ángulo va subiendo hasta llegar a los 150º aproximadamente.

Aunque en los experimentos anteriores se haya tomado una frecuencia de muestreo muy alta capaz de seguir movimientos muy bruscos, de implementarse en un robot no sería necesaria tanta frecuencia. De este modo, en el momento de implementarlo en el simulador y el robot pasaremos de 30 a 3 fps. Por este motivo y para paliar los problemas surgidos en los experimentos anteriores, aparte de reducir la frecuencia de muestreo podríamos pasar los datos por un filtro de paso bajo, de este modo se atenuarían las altas frecuencias correspondientes a las variaciones bruscas y obtendríamos una gráfica con cambios más suaves y que se asemejara más a una curva.

Para facilitar el proceso de alterar la frecuencia hemos incluido un apartado en el código para valorar solo parte de las imágenes que captura.

Para ello valoraremos el resto de dividir la frecuencia de muestreo máxima por la cantidad en la que queremos disminuir la frecuencia (a la mitad, tercera parte, la décima parte, etc) como se aprecia en la figura 67.

```
    USHORT depth;
    contador2++;
    contador3=contador2%10;
    if(contador3==0){

for (i = 0; i < NUI_SKELETON_POSITION_COUNT; i++)
{
```

Figura 67. Código para variar la frecuencia de muestreo

Para llevar a cabo el experimento con filtro se han cogido los datos del movimiento del codo representándolos en una gráfica como muestra la figura 68 y, posteriormente, a dicho movimiento se le ha aplicado un filtro para tratar de eliminar parte de las mediciones "erróneas" y en general el ruido de nuestro movimiento.

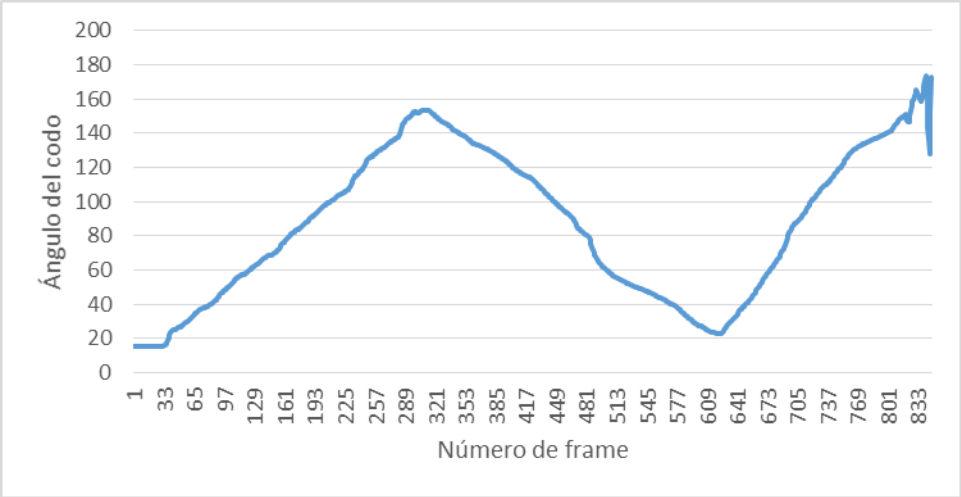

Figura 68. Movimiento del codo sin filtro

Como podemos ver la gráfica de la figura 69 se ha suavizado, sobretodo en puntos en los que solo era una medición la que distorsionaba la gráfica.

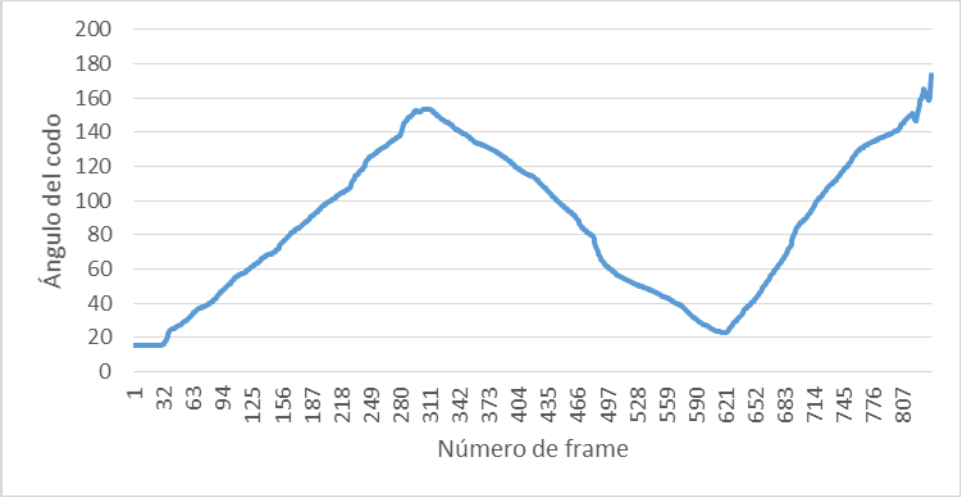

Figura 69. Movimiento del codo con filtro

Para la siguiente prueba representada en la figura 70 bajaremos la frecuencia de muestreo a una décima parte de lo que era anteriormente. En este caso damos menos importancia a puntos en los que hay un salto excesivo entre dos mediciones pero dependes de la suerte para que, la medición no coincida con un dato erróneo, ya que en dicho caso este obtendrá mayor peso.

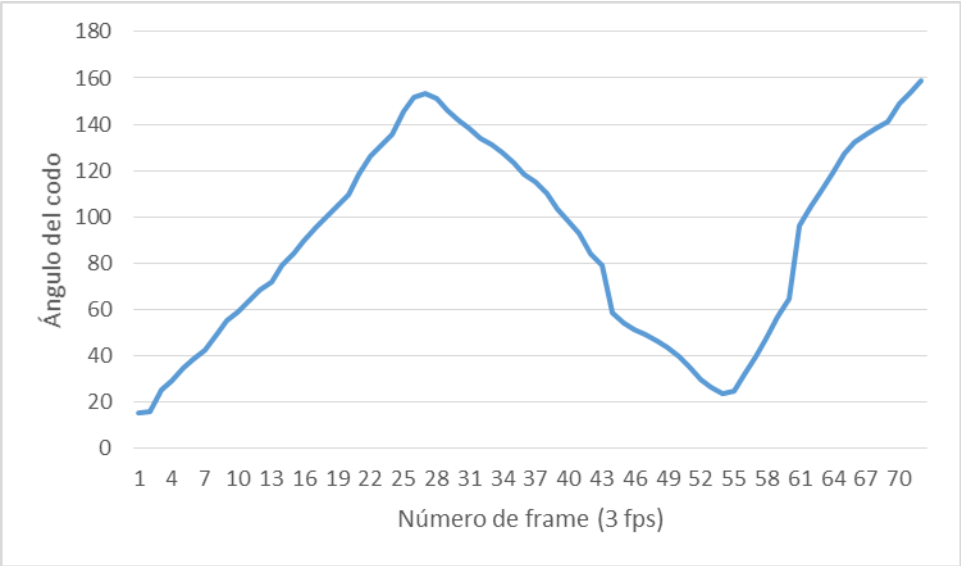

Figura 70. Movimiento del codo a 3 fps

Por último hemos superpuesto los dos métodos en la figura 71. El movimiento representado no será tan fiel al descrito por el brazo y para movimientos muy rápidos no será muy exacto pero por el contrario es capaz de

captar lo más importante del movimiento y dar una salida con variaciones muy constantes perfectas para implementarlas en un robot.

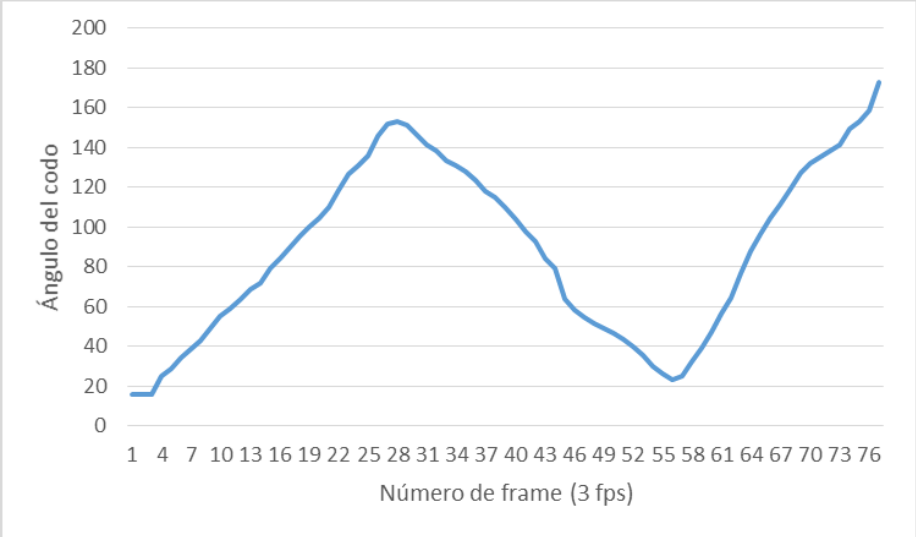

Figura 71. Movimiento del codo a 3 fps y con filtro

A continuación haremos una prueba con un movimiento similar pero mucho más rápido y esta vez sin forzar al software a suponer posiciones. Para este experimento mantendremos la frecuencia de muestreo al máximo, es decir, a 30 fps con el fin de comprobar la capacidad de seguimiento del programa a altas velocidades.

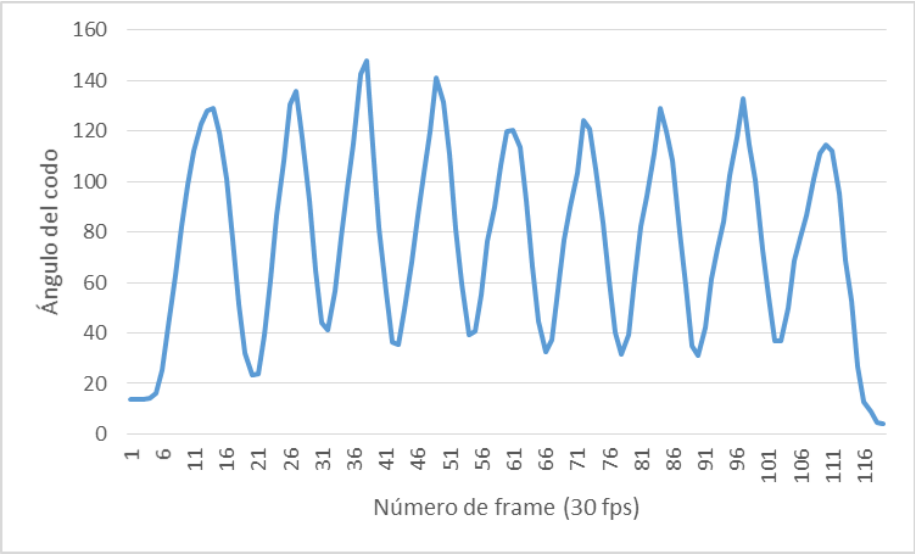

Figura 72. Movimiento rápido del codo

Como se aprecia en la figura 72, el programa es capaz de seguir el movimiento del brazo fielmente aunque aparecen pequeños saltos o escalones. Para paliar estos errores lo someteremos a un filtro de paso bajo.

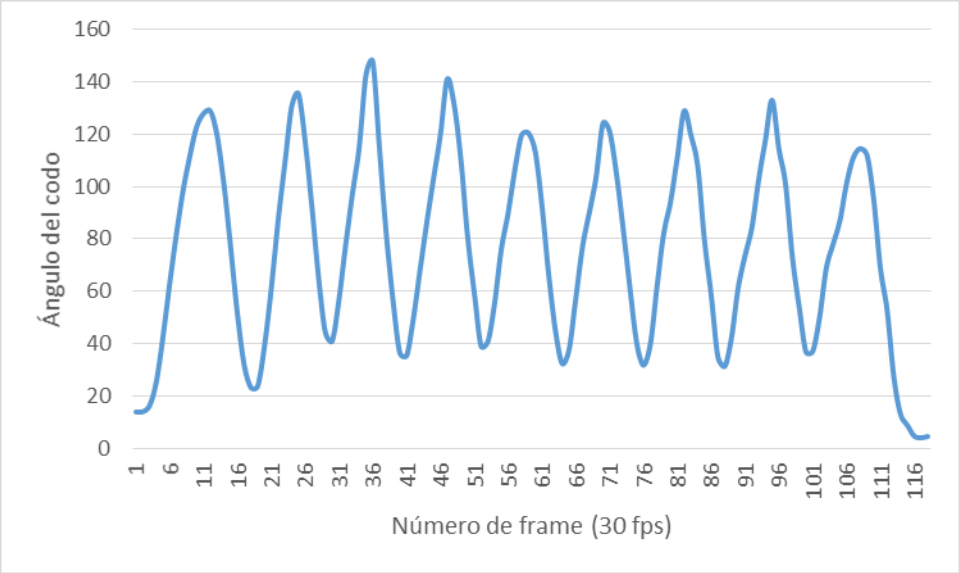

Figura 73. Movimiento rápido del codo con filtro

En la figura 73 se aprecia que las anomalías antes descritas se mitigan dejando el movimiento mucho más constante y suave más favorable para el movimiento del robot.

7. Conclusiones y trabajos futuros

La principal conclusión obtenida tras la realización del proyecto es la cantidad de oportunidades que ofrece la teleoperación de robots humanoides, como su aplicación en todo tipo de trabajos de riesgo o su uso en el ámbito de la asistencia social a ancianos y discapacitados.

El desarrollo del proyecto también ha servido para descubrir las posibilidades de sensores como Kinect para llevar a cabo todo tipo de funciones relacionadas con la visión artificial. Sus capacidades no solo ofrecen buenos resultados a la hora de capturar a un usuario, también es capaz de funcionar como sensor de visión en cualquier robot que necesite captar y analizar su entorno. Además, otro punto a favor del sensor Kinect es la posibilidad del uso del SDK de Microsoft si trabajas en Windows y no te hace falta hacer el tracking de más de dos personas al mismo tiempo debido a su fiabilidad y rapidez a la hora de captar al usuario, así como su facilidad de uso y la gran cantidad de documentación y ejemplos, lo que facilita mucho el aprendizaje.

Respecto al desarrollo del proyecto propiamente dicho, el comienzo consistió en una primera toma de contacto con la visión artificial y en qué consistía realizar el tracking de una persona, pasando posteriormente a trabajar con el dispositivo Kinect, aprendiendo sobre sus posibilidades y funcionamiento gracias a los ejemplos facilitados por *Kinect for Windows SDK*. Gracias a uno de los ejemplos facilitados llamado *SkeletalViewer* y que posteriormente se usó como basa para desarrollar nuestro programa, se alcanzaron los objetivos de detectar a una persona mediante el dispositivo Kinect y realizar su esqueletización.

Gracias a la información obtenida de internet y de otros proyectos que versaban también sobre el uso del dispositivo Kinect para visión artificial, conocer la tecnología disponible en el mercado (programas, drivers, librerías, etc.) así como la que se debería utilizar en el proyecto no fue un paso excesivamente complicado, aunque sí lo fue su posterior instalación, sobre todo la de aquellos programas que debían ser instalados en el sistema operativo Linux, debido a mi falta de experiencia con dicho sistema operativo, aunque esto sirvió para conocer mejor su funcionamiento y ampliar mis conocimientos sobre él.

La parte más difícil fue el cálculo de los ángulos articulares ya que en un principio se decidió utilizar un método bastante intuitivo conocido como RPY (Roll Pitch Yaw) pero, una vez que se conectó el simulador y se probó el programa en él se comprobó que no era el método más adecuado para el

control del brazo por lo que, tras la defensa de este proyecto, se continuará trabajando en este punto hasta conseguir un control fiable y preciso del brazo robótico. Por este motivo se considera solo parcialmente cumplido el objetivo de calcular los ángulos articulares para que el robot realice los movimientos.

Por otro lado, el conseguir simular el robot real y poder realizar pruebas con él fue muy importante ya que demostró por qué se usan simuladores para probar los programas antes de implementarlos en el mundo real. Debido a que el programa no funcionaba como se esperaba, si se hubiese implementado directamente en el robot real habríamos corrido el riesgo de estropear alguna pieza del mismo, mientras que en el simulador se pudieron hacer numerosas pruebas sin ningún riesgo y comprobar que errores se estaban produciendo. Con este paso se consideraron alcanzados los objetivos de conectar el ordenador que ejecutaba el programa con, en este caso, el que ejecutaba el simulador, pasar los ángulos articulares de un ordenador a otro y mostrarlos por pantalla y, por último, usar el simulador para comprobar el funcionamiento del programa.

Después de haber concluido el proyecto con éxito algunas de las posibilidades que nos encontramos como futuras líneas de actuación a la hora de mejorar y ampliar este proyecto son:

- Extender el tracking realizado con el brazo derecho a otras partes del cuerpo. El primer paso y más obvio sería realizar el tracking del brazo izquierdo, lo cual sería bastante fácil basándose en el código ya escrito y daría al robot una mayor versatilidad y capacidad de operación.
  Posteriormente y ya que contamos con un simulador que incluye el robot completo se podría realizar el tracking de cabeza, torso, cadera y piernas, aunque este paso sería un poco más costoso debido a que se tendría que escribir nuevo código al no poder únicamente modificar el realizado para los brazos. Hay que decir que realizar el tracking de las piernas no implica la capacidad de caminar del robot humanoide, al que habría que incluir muchos más sensores y analizar su dinámica ya que, aunque imitara a la perfección el movimiento de la persona, ni sus medidas ni sus pesos son los mismos.

- Otra de las posibilidades de mejora sería aplicar el programa a brazos robóticos con fines industriales o de asistencia a personas. En este caso se debería mejorar el software para una mayor precisión en los movimientos y un mayor control. Además sería bueno aumentar el nivel de telepresencia del usuario mediante diferentes sensores y cámaras.

Universidad Carlos III de Madrid

8. Bibliografía

[1] Información de Visión artificial: http://es.wikipedia.org/wiki/Visi%C3%B3n_artificial.

[2] Información del dispositivo Kinect: http://es.wikipedia.org/wiki/Kinect.

[3] Padilla Montiel, Oscar. *Manipulador teleoperado inalámbricamente.* [Cholula, Puebla, México]: Departamento de Computación, Electrónica y Mecatrónica de la Universidad de las Américas Puebla, Mayo de 2008. http://catarina.udlap.mx/u_dl_a/tales/documentos/lmt/padilla_m_o/.

[4] Información de OpenRAVE: http://en.wikipedia.org/wiki/OpenRAVE.

[5] Información sobre *Kinect for Windows SDK*: http://www.microsoft.com/en-us/kinectforwindows/.

[6] Información del funcionamiento interno del SkeletalViewer de Microsoft: http://es.scribd.com/doc/94544035/SkeletalViewer-Walkthrough.

[7] Magro Viforcos, Eric. *Aplicación de las cámaras 3D al reconocimiento de actividades*. [Leganes, España]: Departamento de Ingeniería de Sistemas y Automática de la Universidad Carlos III, Septiembre 2012. http://e-archivo.uc3m.es/handle/10016/16851.

[8] Alfaro Ballesteros, Santiago. *Sistema de teleoperación mediante una interfaz natural de usuario.* [Leganés, España]: Departamento de Informática de la Universidad Carlos III, Octubre de 2012. http://orff.uc3m.es/handle/10016/16682

[9] Información y ejemplos de Yarp: http://eris.liralab.it/yarpdoc/index.html.

[10] Código a partir del cual se creó el programa *testTEO.* https://svn.code.sf.net/p/roboticslab/code/asibot/example/cpp/testRemoteRaveBot.cpp.

[11] Información sobre las librerías utilizadas en el programa *testTEO*. http://roboticslab.sourceforge.net/asibot/group__asibot__libraries.html

[12] Información sobre CMake. http://www.cmake.org/; http://es.wikipedia.org/wiki/CMake.

[13] Información sobre ángulos de Euler y RPY. http://web.mit.edu/course/2/2.05/www/Handout/HO2.PDF

Universidad Carlos III de Madrid

[14] Información sobre la cadena cinemática de un brazo robótico aplicando ángulos de Euler. http://personal.us.es/jcortes/Material/Material_archivos/Articulos%20PDF/CadenaBrazo.pdf

Nota: Todas las páginas web se encontraban disponibles a fecha del 24 de Septiembre de 2014.

Universidad Carlos III de Madrid

9. Anexos

En este punto se incluirá el código que ha sido programado para el proyecto y el NuiImpl.cpp al completo. No así el conjunto del código que es necesario para el proyecto ya que este se basa el proyecto más extenso SkeletalViewer creado por Microsoft.

Universidad Carlos III de Madrid

A. Código para tracking

Estos fragmentos de código estarán repartidos por el proyecto SkeletalViewerYarp.

SkeletalViewer.cpp

```
    // Create main application window
    HWND hWndApp = CreateDialogParam(
        hInstance,
        MAKEINTRESOURCE(IDD_APP),
        NULL,
        (DLGPROC) CSkeletalViewerApp::MessageRouter,
        reinterpret_cast<LPARAM>(&g_skeletalViewerApp));

    // Show window
    ShowWindow(hWndApp,nCmdShow);
```

```
     ////////////////////ABRIR YARP//////////////////////////

     LPCWSTR szAppTitle(_T("Note"));
     Network yarp;


     if (!yarp.checkNetwork()) {
          LPCWSTR szContents = _T("Failed to find YARP
server.");
          MessageBox( NULL, szContents, szAppTitle, MB_OK |
MB_ICONHAND);

     }

     ////////////////////////////////////////////////////////
```

NuiImpl.cpp

```
//------------------------------------------------------------
----------------
// <copyright file="NuiImpl.cpp" company="Microsoft">
//         Copyright (c) Microsoft Corporation.   All rights
reserved.
```

```
// </copyright>
//------------------------------------------------------------------------------

// Implementation of CSkeletalViewerApp methods dealing with NUI processing

#include "stdafx.h"
#include "SkeletalViewer.h"
#include "resource.h"
#include <mmsystem.h>
#include <assert.h>
#include <strsafe.h>
#include <math.h>
#include <yarp/os/Network.h>
#include <yarp/os/Port.h>
#include <yarp/os/Bottle.h>
#include <yarp/os/all.h>

using namespace yarp::os;

float X=0, Y=0, Z=0, X1=0, Y1=0, Z1=0, posicionx=-100, posiciony=-100, posicionz=-100;
float hombrox=0, hombroy=0, hombroz=0, codox=0, codoy=0, codoz=0;
float angulo_hombro_xy=0, angulo_hombro_xz=0, angulo_hombro_yz=0, angulo_codo_xy=0, angulo_codo_xz=0, angulo_codo_yz=0;
float angulo_codo_final=0, angulo_centro_hombro_xy=0, angulo_centro_hombro_xz=0, angulo_centro_hombro_yz=0;
float Ax=0, Ay=0, Az=0, Axy=0, Axz=0, Ayz=0, Bx=0, By=0, Bz=0, Bxy=0, Bxz=0, Byz=0,Cxy=0, Cxz=0, Cyz=0, Cx=0, Cy=0, Cz=0;
float Dxy=0, Dxz=0, Dyz=0, Dx=0, Dy=0, Dz=0, Centrox=0, Centroy=0, Centroz=0, Exy=0, Exz=0, Eyz=0, Ex=0, Ey=0, Ez=0;
float alfa=0, beta=0, gamma=0, m_xy=0, n_xy=0, y=0, m_xz=0, n_xz=0, z=0, m_yz=0, n_yz=0, y2=0;
float r11=0, r12=0, r13=0, r21=0, r22=0, r23=0, r31=0, r32=0, r33=0;
int primer=1;

FILE *doc;

Port output;
```

```
//-----------------------------------------------------------------
-----------------

doc = fopen("prueba.txt","a");

if(primer==1){
      output.open("/sender");
      primer=2;
}

//-----------------------------------------------------------------
-----------------


for (i = 0; i < NUI_SKELETON_POSITION_COUNT; i++)
{

int contador=1;
if((contador<=2000)&&((i==8)||(i==9)||(i==10)||(i==2)))
{

 posicionx=pSkel->SkeletonPositions[i].x;
 posiciony=pSkel->SkeletonPositions[i].y;
 posicionz=pSkel->SkeletonPositions[i].z;

  if(i==2)
  {
   Centrox=posicionx;
   Centroy=posiciony;
   Centroz=posicionz;
  }

  else
   if(i==8)
   {
    hombrox=posicionx;
    hombroy=posiciony;
    hombroz=posicionz;

    Dx=hombrox-Centrox;
    Dy=hombroy-Centroy;
    Dz=hombroz-Centroz;

    Dxy=sqrt((Dx*Dx)+(Dy*Dy));//para plano XY
    m_xy=(Centroy-hombroy)/(Centrox-hombrox);
```

```
    n_xy=Centroy-Centrox*m_xy;
    angulo_centro_hombro_xy=(atan((Centroy-hombroy)/(hombrox-
Centrox)))*(180/3.1416);//para calcular el angulo formado por el
centro de los     hombros y el hombro derecho

    Dxz=sqrt((Dx*Dx)+(Dz*Dz));//para plano XZ
    m_xz=(Centroz-hombroz)/(Centrox-hombrox);
    n_xz=Centroz-Centrox*m_xz;
    angulo_centro_hombro_xz=(atan((Centroz-hombroz)/(hombrox-
Centrox)))*(180/3.1416);

    Dyz=sqrt((Dy*Dy)+(Dz*Dz));//para plano YZ
    m_yz=(Centroy-hombroy)/(Centroz-hombroz);
    n_yz=Centroy-Centroz*m_yz;
    angulo_centro_hombro_yz=(atan((hombroz-Centroz)/(Centroy-
hombroy)))*(180/3.1416);

    }

    else
     if(i==9)
     {

      codox=posicionx;
      codoy=posiciony;
      codoz=posicionz;

      Ax=codox-hombrox;
      Ay=codoy-hombroy;
      Az=codoz-hombroz;

      Axy=sqrt((Ax*Ax)+(Ay*Ay));//para plano XY
      Axz=sqrt((Ax*Ax)+(Az*Az));//para plano XZ
      Ayz=sqrt((Az*Az)+(Ay*Ay));//para plano YZ

      Ex=codox-Centrox;
      Ey=codoy-Centroy;
      Ez=codoz-Centroz;

      Exy=sqrt((Ex*Ex)+(Ey*Ey));//para plano XY
      Exz=sqrt((Ex*Ex)+(Ez*Ez));//para plano XZ
      Eyz=sqrt((Ez*Ez)+(Ey*Ey));//para plano YZ
```

```
      angulo_hombro_xy=(acos((Axy*Axy+Dxy*Dxy-
Exy*Exy)/(2*Axy*Dxy)))*(180/3.1416);
      angulo_hombro_xz=(acos((Axz*Axz+Dxz*Dxz-
Exz*Exz)/(2*Axz*Dxz)))*(180/3.1416);
      angulo_hombro_yz=(acos((Ayz*Ayz+Dyz*Dyz-
Eyz*Eyz)/(2*Ayz*Dyz)))*(180/3.1416);

      y=codox*m_xy+n_xy;//posición   del   codo   si   estuviese
contenido en la recta
      z=codox*m_xz+n_xz;
      y2=codoz*m_yz+n_yz;

     if(y<=codoy)
     {

      angulo_hombro_xy=270-angulo_hombro_xy-
angulo_centro_hombro_xy;

     }
     else
      if(y>codoy)
      {

       angulo_hombro_xy=angulo_hombro_xy-90-
angulo_centro_hombro_xy;

      }
     if(z<=codoz) //el codo esta por detras de la linea que une
centro y hombro
     {

      angulo_hombro_xz=270-angulo_hombro_xz-
angulo_centro_hombro_xz;

     }

     else
      if(z>codoz) //el codo esta por delante de la linea que une
centro-hombro
      {

       angulo_hombro_xz=angulo_hombro_xz-90-
     angulo_centro_hombro_xz;
```

```
        if(angulo_hombro_xz<0)
        {

         angulo_hombro_xz=0;

         }
        }

      if(codoz>hombroz)
      {

       if((y2>codoy)&&(Centroz<hombroz))
       {

        angulo_hombro_yz=360+(180-angulo_hombro_yz-
angulo_centro_hombro_yz);

       }

       else
        if((y2<codoy)&&(m_yz>0))
        {

         angulo_hombro_yz=360-
(180+angulo_hombro_yz+angulo_centro_hombro_yz);

        }
        else
        {

         angulo_hombro_yz=360-(180-
angulo_hombro_yz+angulo_centro_hombro_yz);

        }
      }
      else
       if(codoz<=hombroz)
       {
        if((y2>codoy)&&(Centroz>hombroz))
        {

         angulo_hombro_yz=-(180-
angulo_hombro_yz+angulo_centro_hombro_yz);
```

```
        }
        else
         if((y2<codoy)&&(m_yz<0))
         {

          angulo_hombro_yz=180-
angulo_centro_hombro_yz+angulo_hombro_yz;

         }
         else
         {

          angulo_hombro_yz=180-angulo_hombro_yz-
angulo_centro_hombro_yz;

         }
        }
      }
      else
      if(i==10)
      {

       Bx=posicionx-codox;
       By=posiciony-codoy;
       Bz=posicionz-codoz;

       Bxy=sqrt((Bx*Bx)+(By*By));
       Bxz=sqrt((Bx*Bx)+(Bz*Bz));
       Byz=sqrt((Bz*Bz)+(By*By));

       Cx=posicionx-hombrox;
       Cy=posiciony-hombroy;
       Cz=posicionz-hombroz;

       Cxy=sqrt((Cx*Cx)+(Cy*Cy));
       Cxz=sqrt((Cx*Cx)+(Cz*Cz));
       Cyz=sqrt((Cz*Cz)+(Cy*Cy));

       angulo_codo_xy=180-((acos((Axy*Axy+Bxy*Bxy-
Cxy*Cxy)/(2*Axy*Bxy)))*(180/3.1416));
       angulo_codo_xz=180-((acos((Axz*Axz+Bxz*Bxz-
Cxz*Cxz)/(2*Axz*Bxz)))*(180/3.1416));
```

```
      angulo_codo_yz=180-((acos((Ayz*Ayz+Byz*Byz-
Cyz*Cyz)/(2*Ayz*Byz)))*(180/3.1416));

     if(angulo_hombro_xy>180)
     {

      angulo_hombro_xy=angulo_hombro_xy-360;

     }
     if(angulo_hombro_xz>180)
     {

      angulo_hombro_xz=angulo_hombro_xz-360;

     }
     if(angulo_hombro_yz>180)
     {

      angulo_hombro_yz=angulo_hombro_yz-360;

     }
     if(angulo_hombro_xy>90)
     {

      angulo_hombro_yz=90-(angulo_hombro_yz-90);
     }

     if(angulo_hombro_yz<90 && codoy>hombroy)
     {

      angulo_hombro_yz=-angulo_hombro_yz;
     }
     if(codoz<hombroz && codoy<hombroy && angulo_hombro_xy>45 &&
angulo_hombro_yz>45)
     {

      angulo_hombro_xy=angulo_hombro_xy-angulo_hombro_yz;

     }

     if(angulo_hombro_xz>45)

     {
```

```
     angulo_codo_final=angulo_codo_xy;
     }
      Else
      {

       angulo_codo_final=angulo_codo_yz;

      }

//////////////////////////////YARP//////////////////////////////

  Bottle codo;
  Bottle hombroxy,hombroxz,hombroyz;

  hombroxy.addDouble(angulo_hombro_xy);
  output.write(hombroxy);//send the message

  hombroxz.addDouble(angulo_hombro_xz);
  output.write(hombroxz);

  hombroyz.addDouble(angulo_hombro_yz);
  output.write(hombroyz);

  codo.addDouble(angulo_codo_final);
  output.write(codo);


  }
  contador++;


 }

//////////////////////////////////////////////////////////////////

NuiTransformSkeletonToDepthImage(    pSkel->SkeletonPositions[i],
&m_Points[i].x, &m_Points[i].y, &depth );

        m_Points[i].x = (m_Points[i].x * width) / 320;
        m_Points[i].y = (m_Points[i].y * height) / 240;}
```

B. Código de recepción y simulación

```
-*-mode:C++; tab-width:4; c-basic-offset:4; indent-tabs-mode:nil
-*-
/*
* Author: Juan G Victoresand Lorenzo Moral Durán
*
*  Contribs:  Paul  Fitzpatrick  and  Giacomo  Spigler  (YARP
dev/motortest.cpp example)
*
* CopyPolicy: Released under the terms of the LGPLv2.1 or later,
see license/LGPL.TXT
*/

#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <cstring>
#include <string.h>
#include <yarp/os/all.h>
#include <yarp/dev/all.h>
#include <yarp/os/Network.h>
#include <yarp/os/Port.h>
#include <yarp/os/Bottle.h>

using namespace yarp::os;
using namespace yarp::dev;

int main(int argc, char *argv[])
{

 printf("Starting program\n");
 printf("WARNING: requires a running instance of RaveBot (i.e.
testRaveBotor cartesianServer)\n");
 Network yarp;
    if (!Network::checkNetwork())
    {

      printf("Please start a yarp name server first\n");
      return(-1);
    }
    int i=1;
    double hombro=0, codo, dato, cont=1;
    Port input;
```

```
    Property options;
    options.put("device","remote_controlboard");
    options.put("remote","/teoSim/rightArm");
    options.put("local","/local");
    PolyDriver dd(options);
    if(!dd.isValid())
    {

      printf("RaveBot device not available.\n");
      dd.close();
      Network::fini();
      return 1;
    }

    IPositionControl *pos;
    bool ok = dd.view(pos);;
    if (!ok)
    {
     printf("[warning] Problems acquiring robot interface\n");
     return false;
    }
    else
     printf("[success] testAsibot acquired robot interface\n");
     pos->setPositionMode();
     int eje=1;
     input.open("/receiver");
     Network::connect("/sender","/receiver");

     pos->positionMove(0, 0); // se coloca en posición de inicio
     pos->positionMove(1, 0);
     pos->positionMove(2, 0);
     pos->positionMove(3, 0);
     Time::delay(10);

     while(1)
     {
      Bottle bot;
      input.read(bot);
      dato=atof(bot.toString().c_str());//paso   de   string   a
double
      if(cont==1)
      {

       hombro=dato;
```

```
      eje=1;//plano XY
      printf("El angulo del hombro XY es %f\n", hombro);
      printf("test positionMove(%d, %f)\n", eje, hombro);
      //paso de datos al simulador
      pos->positionMove(eje, hombro);
      cont=2;
     }
      else
       if(cont==2)
       {
        hombro=dato;
        eje=2;//plano XZ
        printf("El angulo del hombro XZ es %f\n", hombro);
        printf("test positionMove(%d, %f)\n", eje, hombro);
        pos->positionMove(eje, hombro);
        cont=3;
       }
        else
         if(cont==3)
         {

          hombro=dato;
          eje=0;//plano YZ
          printf("El angulo del hombro YZ es %f\n", hombro);
          printf("test positionMove(%d, %f)\n", eje, hombro);
          pos->positionMove(eje, hombro);
          cont=4;
         }
         else
          if(cont==4)
          {

           codo=dato;
           eje=3;//codo
           printf("El angulo del codo es %f\n", codo);
           printf("test positionMove(%d, %f)\n", eje, codo);
           pos->positionMove(eje, codo);
           cont=1;
          }
         }
         input.close();
         IEncoders *enc;
         ok = dd.view(enc);
         return 0;}
```

Universidad Carlos III de Madrid

C. Presupuesto

Coste laboral

Apellidos y Nombre	Categoría	Horas dedicadas	Coste por hora	Coste
Solano García, Iván	Ingeniero	120	25 €/h	3.000 €
García Haro, Juan Miguel	Ingeniero	15	50 €/h	750 €
			Total	3.750 €

Coste de equipo

Descripción	Coste
Dispositivo Kinect	150 €
Cable de red cruzado	6 €
Total	156 €

Coste total

Coste laboral	3.750 €
Coste de equipo	156 €
TOTAL	3.906 €